

Classes

Amrita Kaur

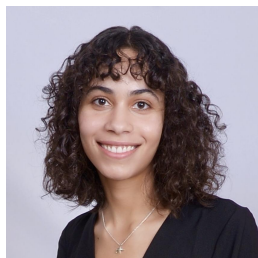
July 20, 2023

Announcements

- None :)

Subsets

Given a group of people, generate all possible teams, or subsets, of these people:



{}

{"Amrita"}

{"Elyse"}

{"Taylor"}

{"Amrita", "Elyse"}

{"Amrita", "Taylor"}

{"Elyse", "Taylor"}

{"Amrita", "Elyse", "Taylor"}

Making a Decision Tree

- Decision at each step (each level of the tree)
 - Are we going to include a given element in our subset?
- Options at each decision (branches from each node)
 - Include the element
 - Don't Include the element
- Information you need to store along the way
 - Set you've built so far
 - Remaining elements in original set

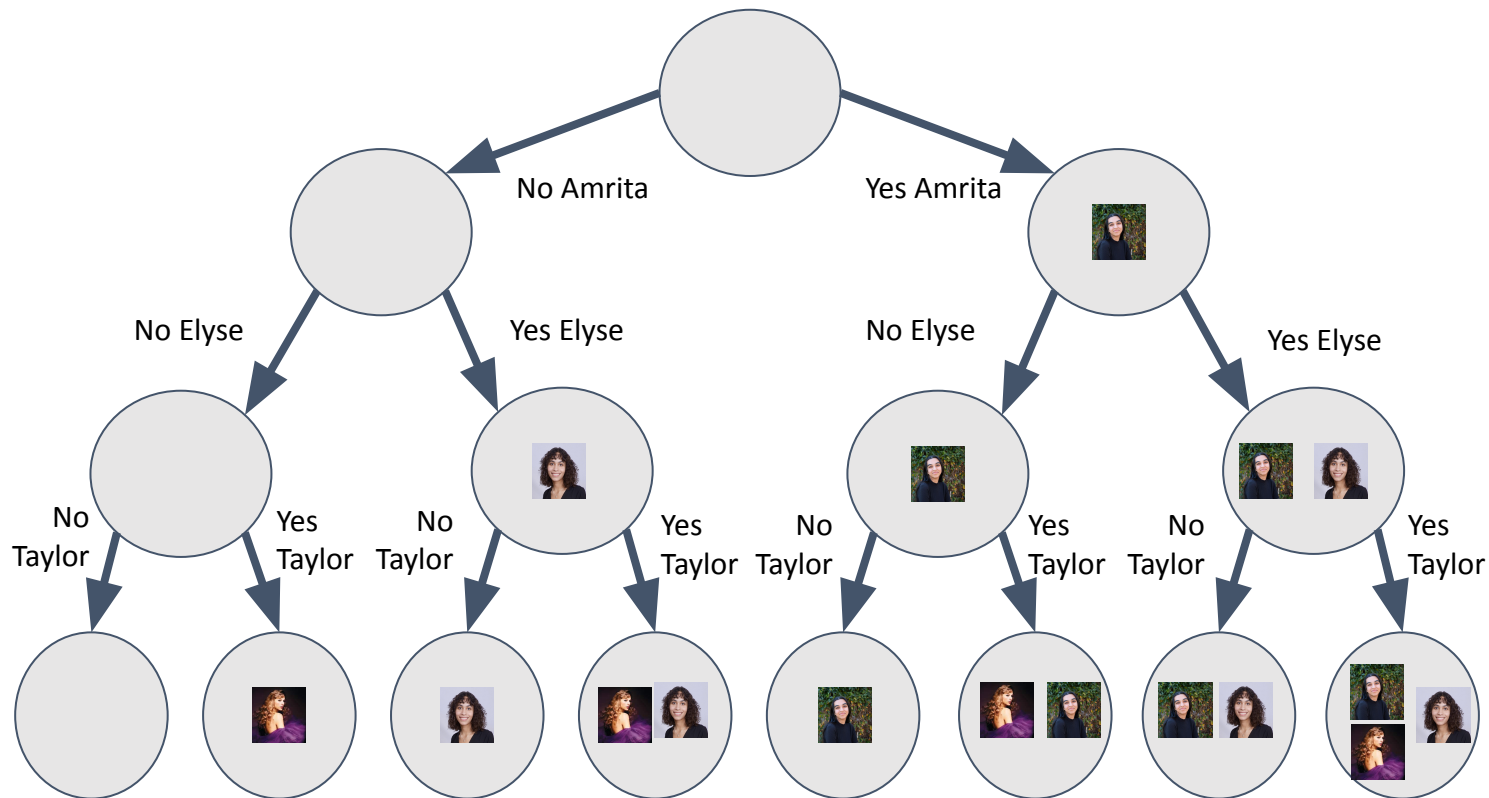
Remaining Elements:

{"Amrita",
"Elyse",
"Taylor"}

{"Elyse",
"Taylor"}

{"Taylor"}

{}



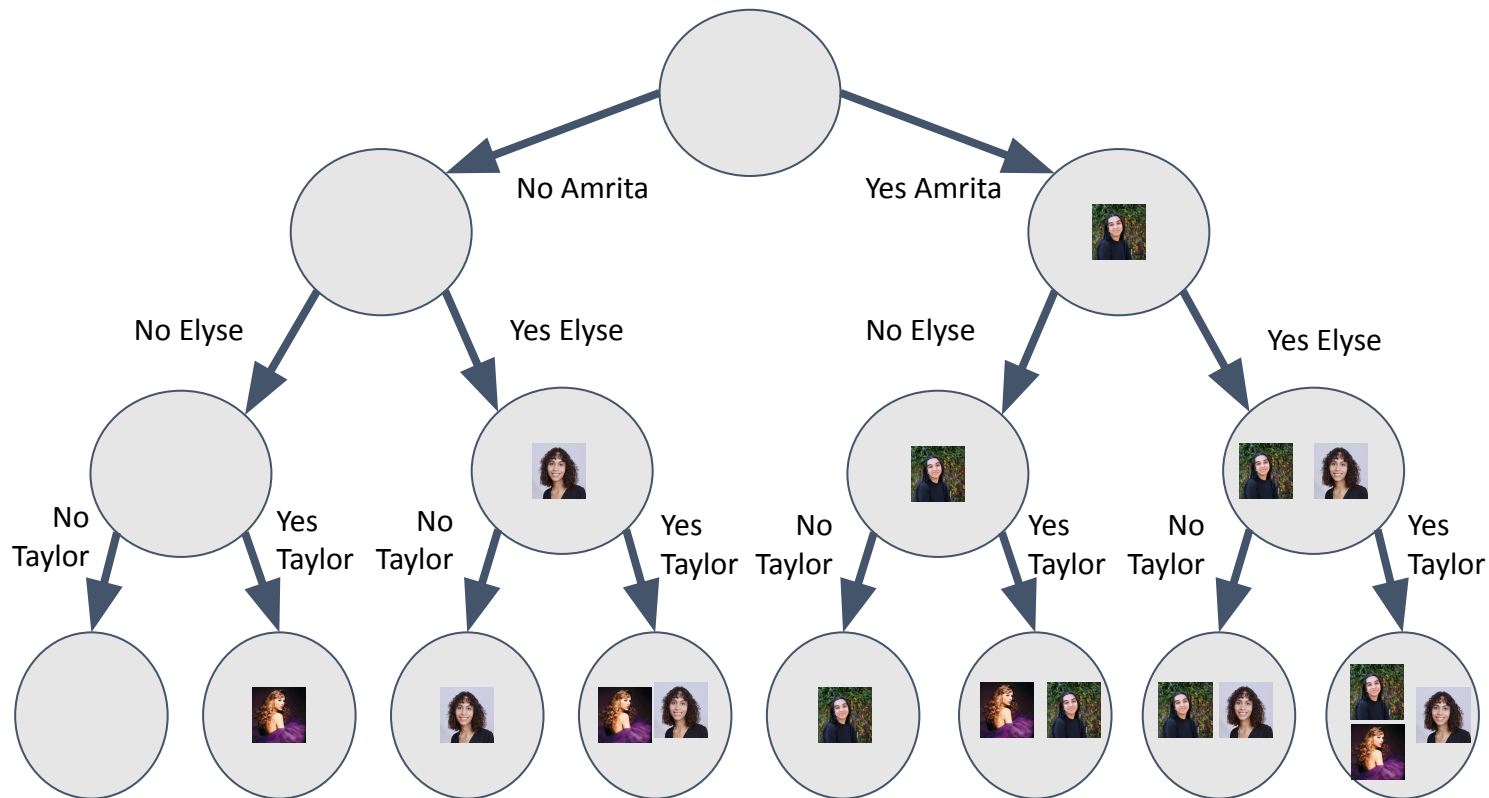
Remaining Elements:

{"Amrita",
"Elyse",
"Taylor"}

{"Elyse",
"Taylor"}

{"Taylor"}

{}



Base Case: No remaining people to choose from

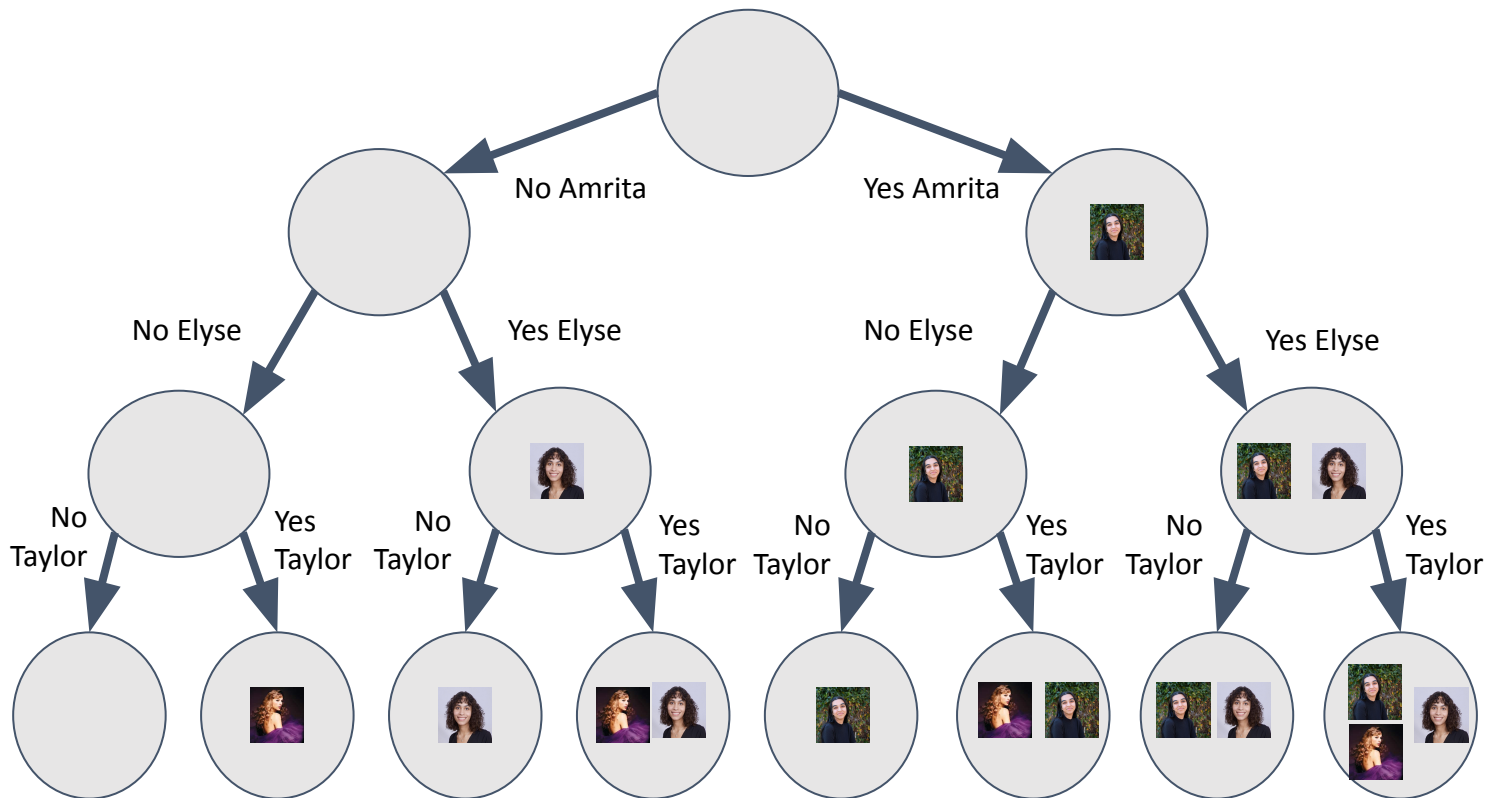
Remaining Elements:

{"Amrita",
"Elyse",
"Taylor"}

{"Elyse",
"Taylor"}

{"Taylor"}

{}



Recursive Case: Pick someone from set. Choose whether to include them.

Takeaways

- "Choose / explore / unchoose" pattern in backtracking

```
// choose
string elem = remaining.first();
remaining = remaining - elem;
// explore
listSubsetHelper(remaining, chosen);
chosen = chosen + elem
listSubsetHelper(remaining, chosen);
// unchoose by adding it back to possible choices
chosen = chosen - elem;
remaining = remaining + elem;
```


Takeaways

- "Choose / explore / unchoose" pattern in backtracking

```
// choose
string elem = remaining.first();
remaining = remaining - elem;
// explore
listSubsetHelper(remaining, chosen); // do not add elem to chosen
chosen = chosen + elem
listSubsetHelper(remaining, chosen);
// unchoose by adding it back to possible choices
chosen = chosen - elem;
remaining = remaining + elem;
```

Takeaways

- "Choose / explore / unchoose" pattern in backtracking

```
// choose
string elem = remaining.first();
remaining = remaining - elem;
// explore
listSubsetHelper(remaining, chosen);
chosen = chosen + elem
listSubsetHelper(remaining, chosen); // add elem to chosen
// unchoose by adding it back to possible choices
chosen = chosen - elem;
remaining = remaining + elem;
```

Takeaways

- "Choose / explore / unchoose" pattern in backtracking
 - Necessary because we're passing sets by reference and editing them

```
// choose
string elem = remaining.first();
remaining = remaining - elem;
// explore
listSubsetHelper(remaining, chosen);
chosen = chosen + elem
listSubsetHelper(remaining, chosen);
// unchoose by adding it back to possible choices
chosen = chosen - elem;
remaining = remaining + elem;
```

Solution Code for Subsets

```
void listSubsetsHelper(Set<string>& remaining, Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

void listSubsets(Set<string>& choices) {
    Set<string> tracked;
    listSubsetsHelper(choices, tracked);
}
```

Input:
choices = {"E", "T"};

What is the output of this function:

- 1) With the unchoose step?
- 2) Without the unchoose step?

Tracing through Broken Code

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()


```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

```
main()  
    friends:  
    {"E", "T"}
```

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()

friends:

{"E", "T"}

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

```
{“E”, “T”}
```

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}


```
void listSubsetsHelper(Set<string>& remaining,  
                      Set<string>& chosen) {  
    if (remaining.isEmpty()) {  
        cout << chosen << endl;  
        return;  
    }  
    string current = remaining.first();  
    remaining = remaining - current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen + current;  
    listSubsetsHelper(remaining, chosen);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

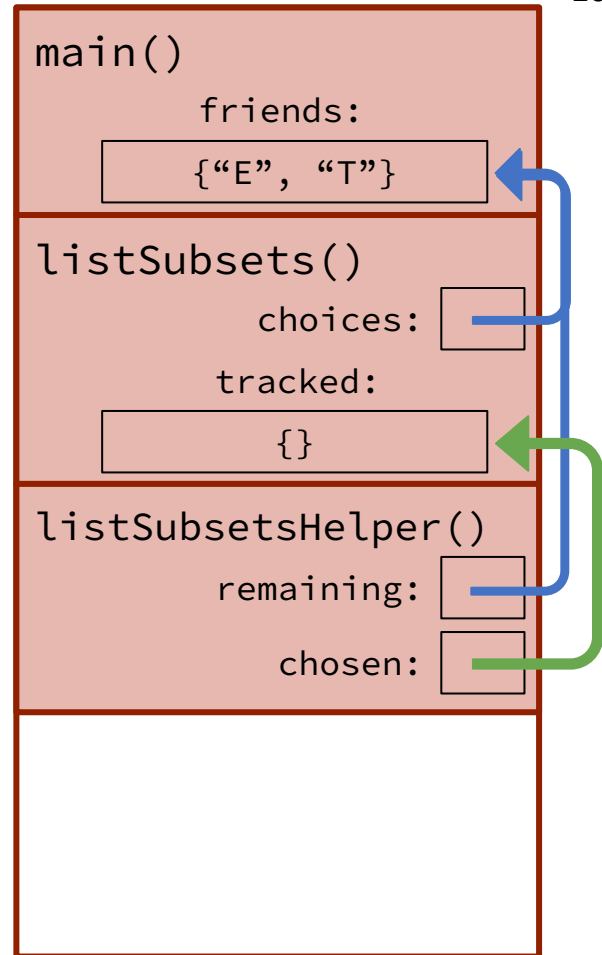
{}

listSubsetsHelper()

remaining:

chosen:

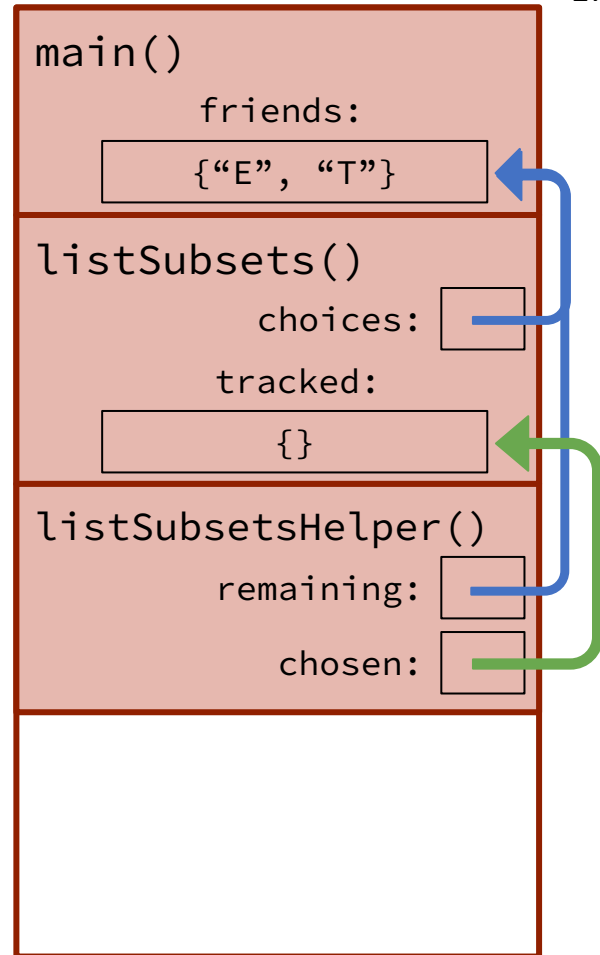
```
void listSubsetsHelper(Set<string>& remaining,  
                      Set<string>& chosen) {  
    if (remaining.isEmpty()) {  
        cout << chosen << endl;  
        return;  
    }  
    string current = remaining.first();  
    remaining = remaining - current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen + current;  
    listSubsetsHelper(remaining, chosen);  
}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```
void listSubsetsHelper(Set<string>& remaining,  
                      Set<string>& chosen) {  
    if (remaining.isEmpty()) {  
        cout << chosen << endl;  
        return;  
    }  
    string current = remaining.first();  
    remaining = remaining - current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen + current;  
    listSubsetsHelper(remaining, chosen);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: “E”

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

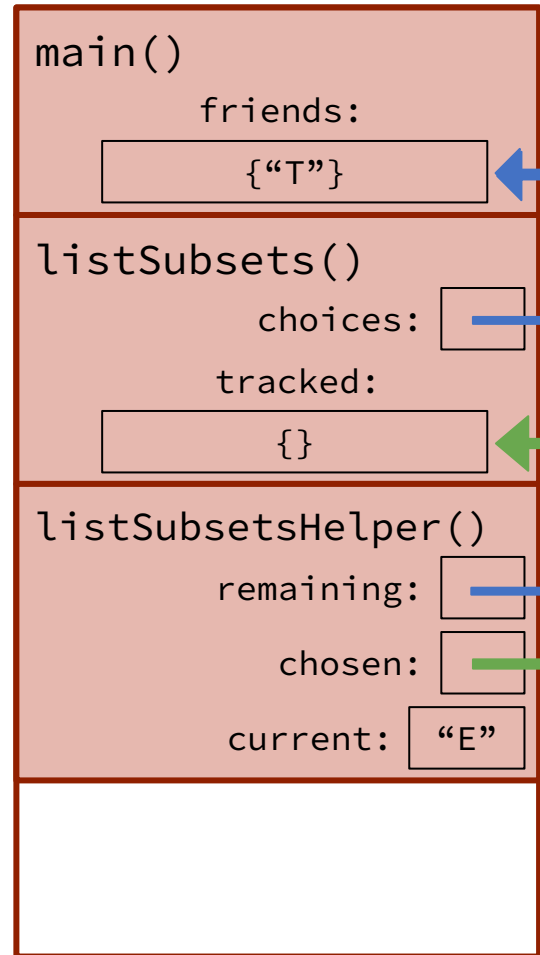
chosen:

current: “E”

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

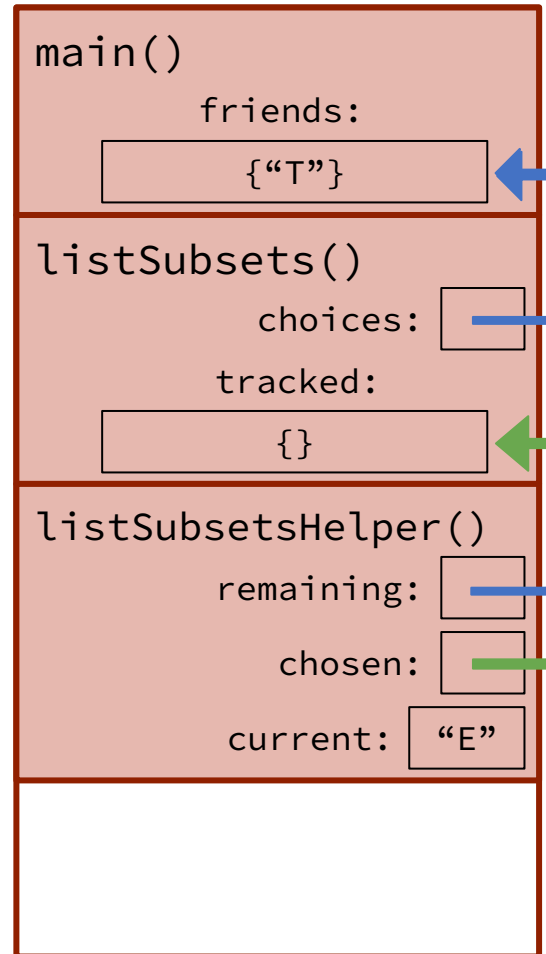
```



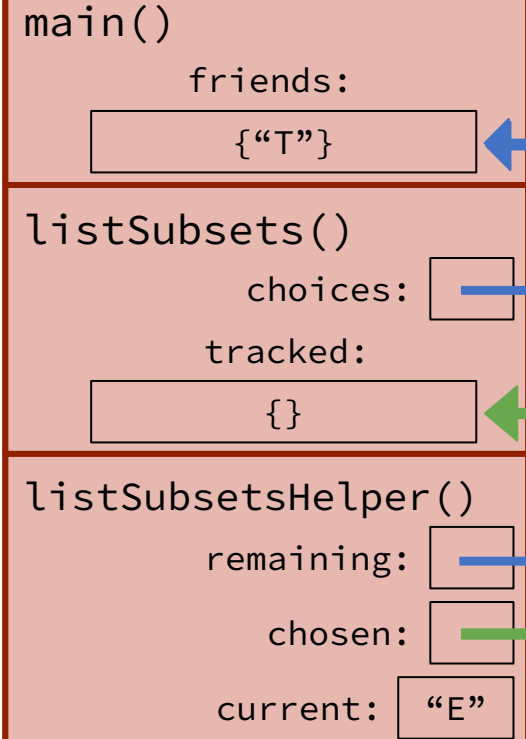
```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```




```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{“T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: “E”

listSubsetsHelper()

remaining:

chosen:

33

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{“T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: “E”

listSubsetsHelper()

remaining:

chosen:

34

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{“T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: “E”

listSubsetsHelper()

remaining:

chosen:

35

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

main()

friends:

{ "T" }

listSubsets()

choices:

tracked:

{ }

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{“T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: “E”

listSubsetsHelper()

remaining:

chosen:

current: “T”

37

iversity

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

38

iversity

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

39

iversity

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

main()

friends:

{}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

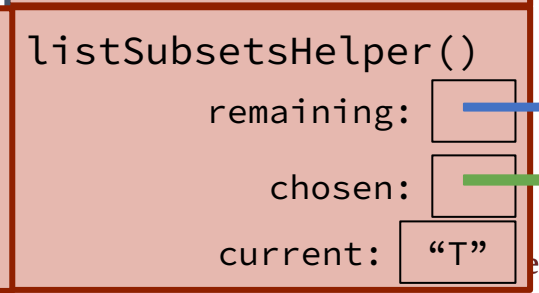
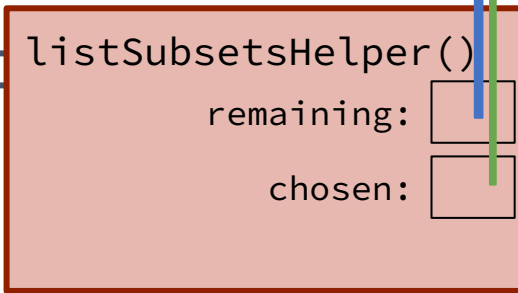
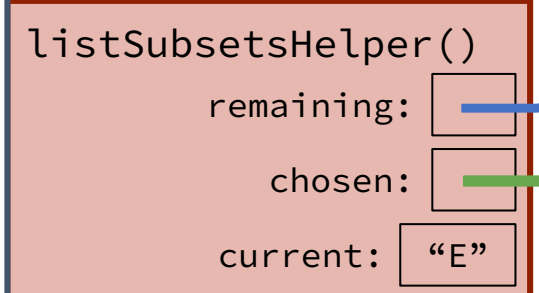
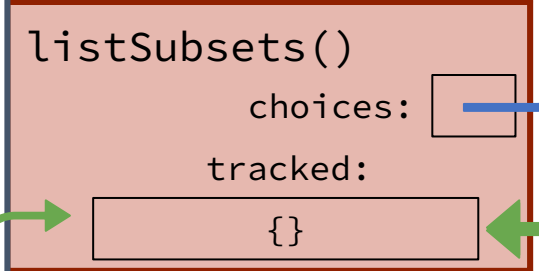
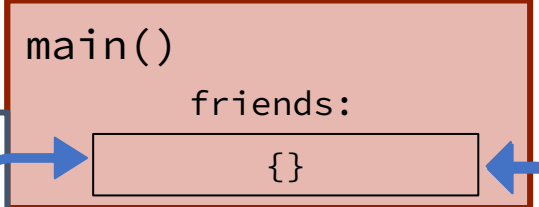
current: "T"

40

iversity


```
void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
```

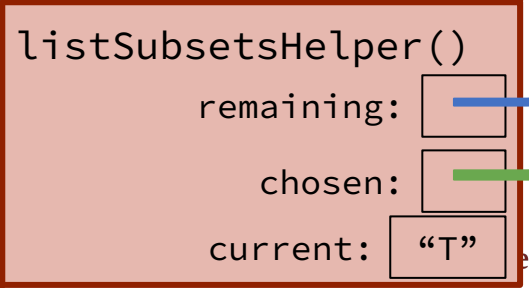
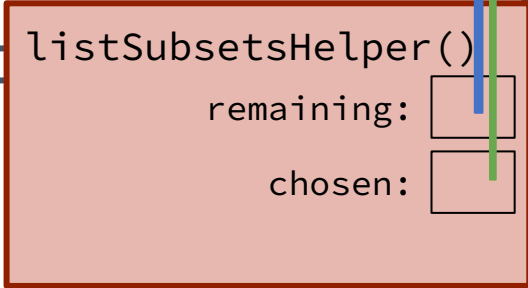
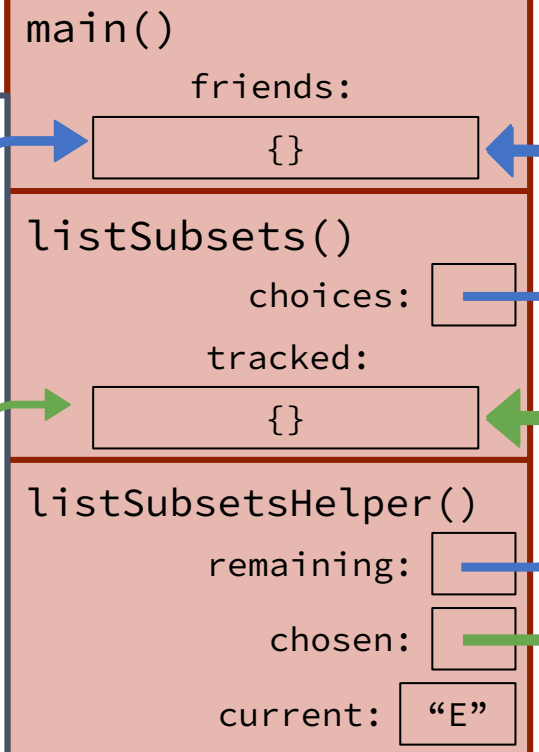
```
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```



```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

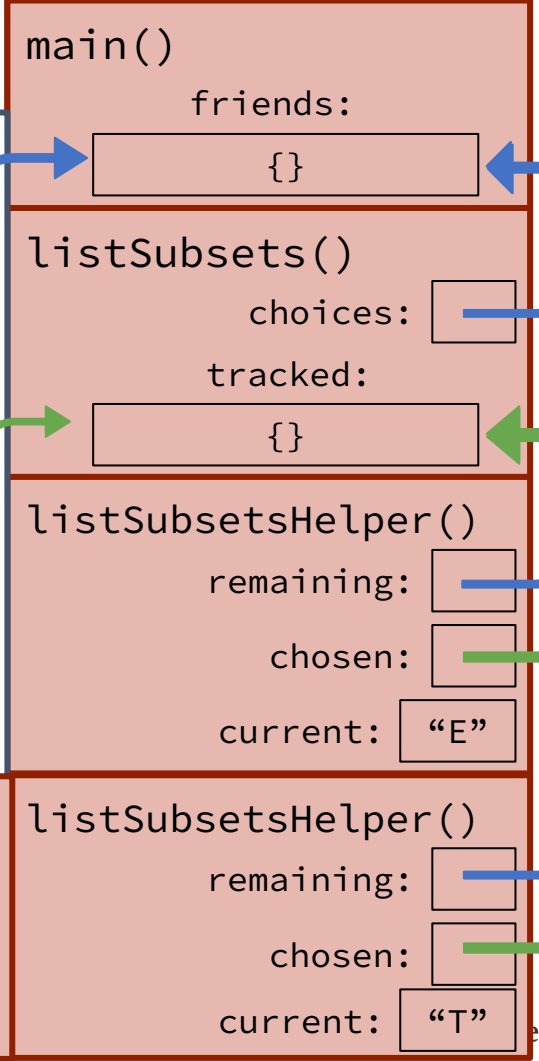
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

{}

listSubsetsHelper()

remaining:

chosen:

listSubsetsHelper()

remaining:

chosen:

current: "T"

main()

friends:

{}

listSubsets()

choices:

tracked:

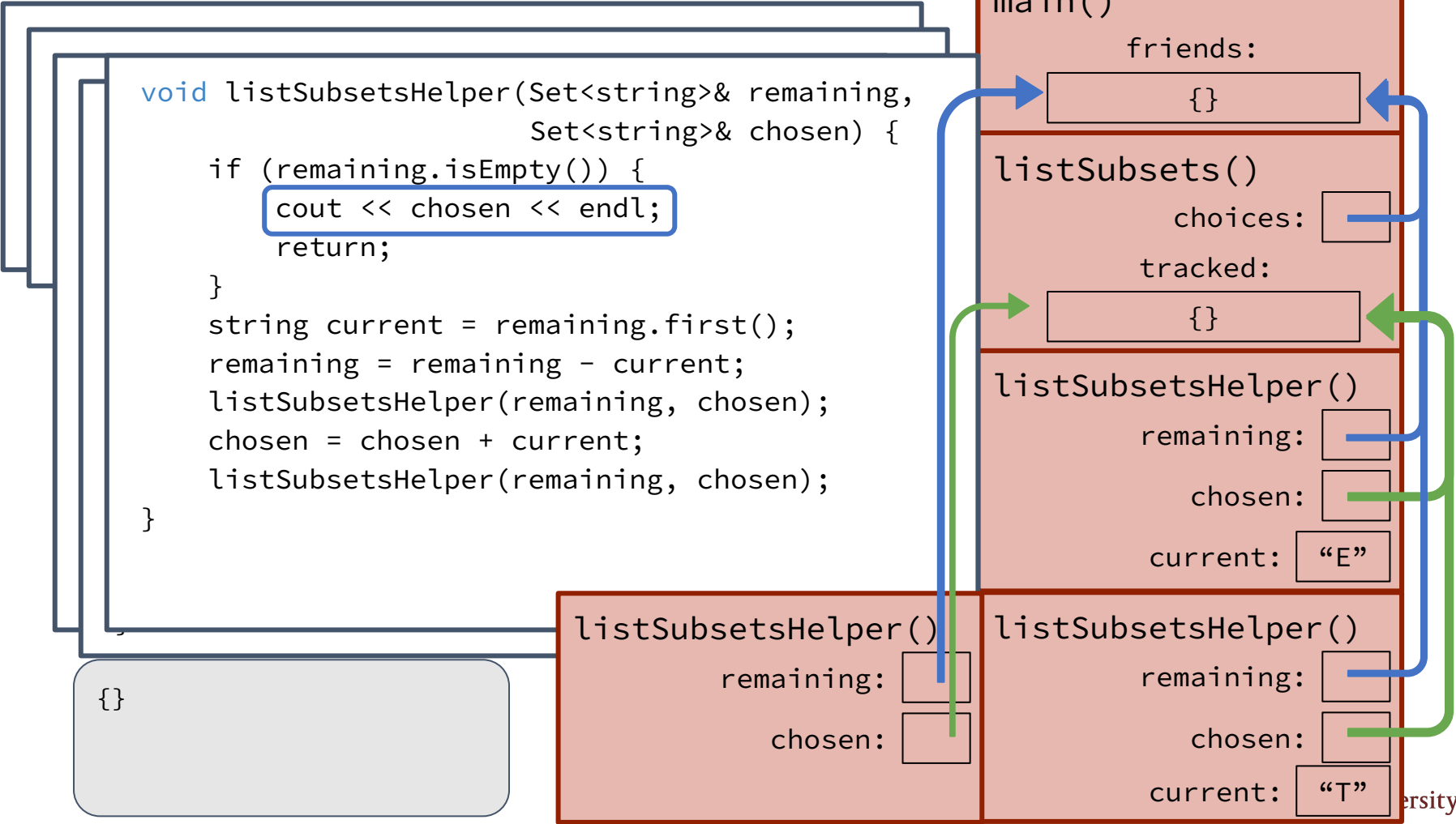
{}

listSubsetsHelper()

remaining:

chosen:

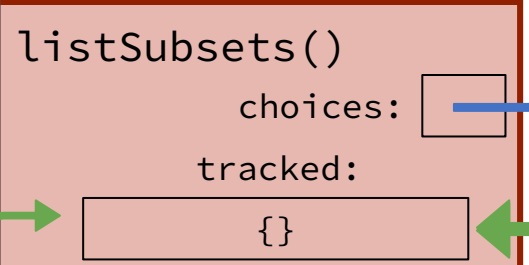
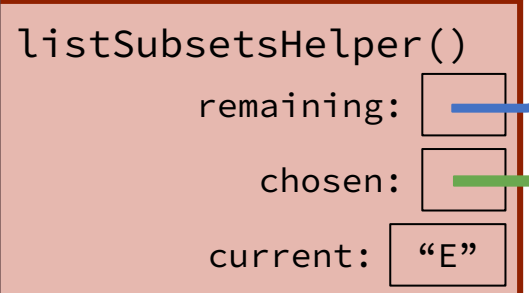
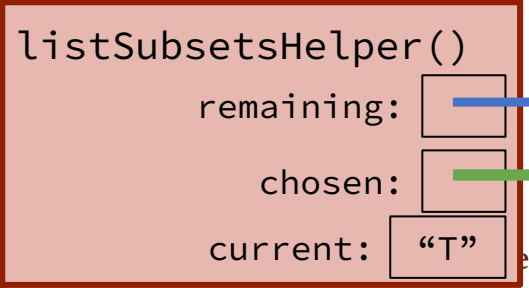
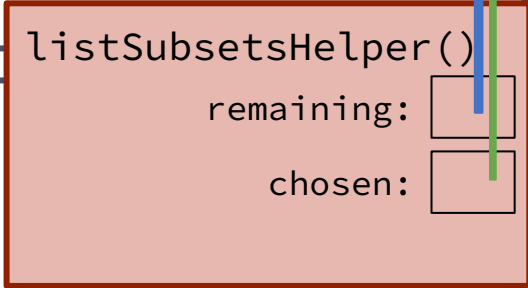
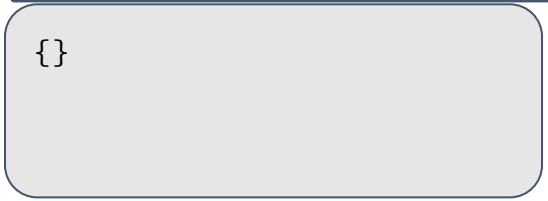
current: "E"



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

```
{}
```

main()

friends:

```
{}
```

listSubsets()

choices:

tracked:

```
{}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

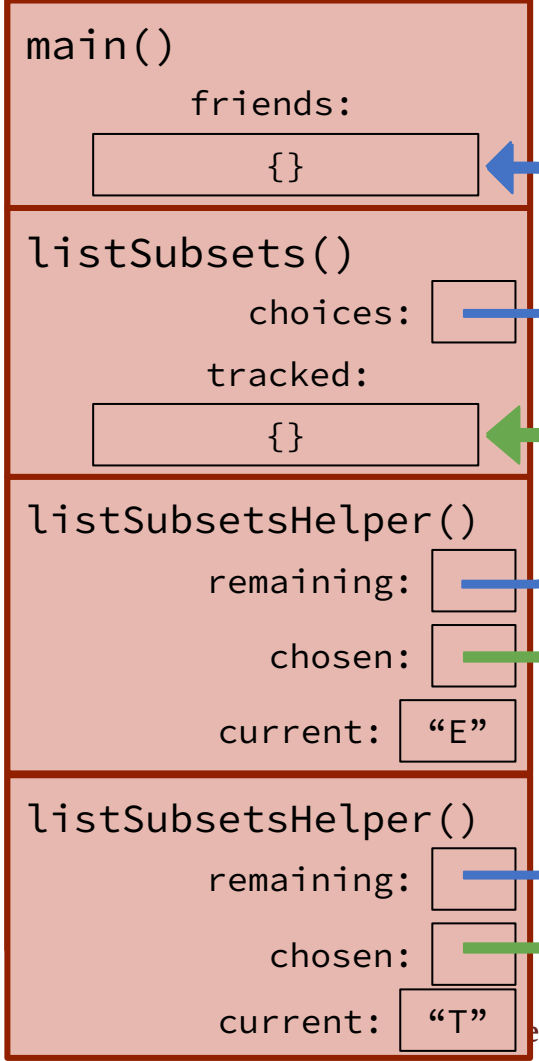
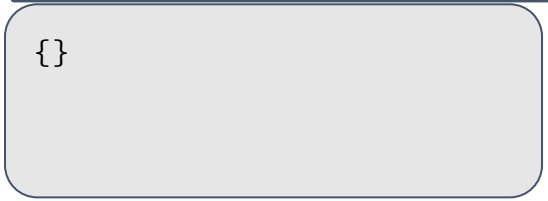
chosen:

current: "T"

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

```
{}
```

main()

friends:

```
{}
```

listSubsets()

choices:

tracked:

```
{"T"}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

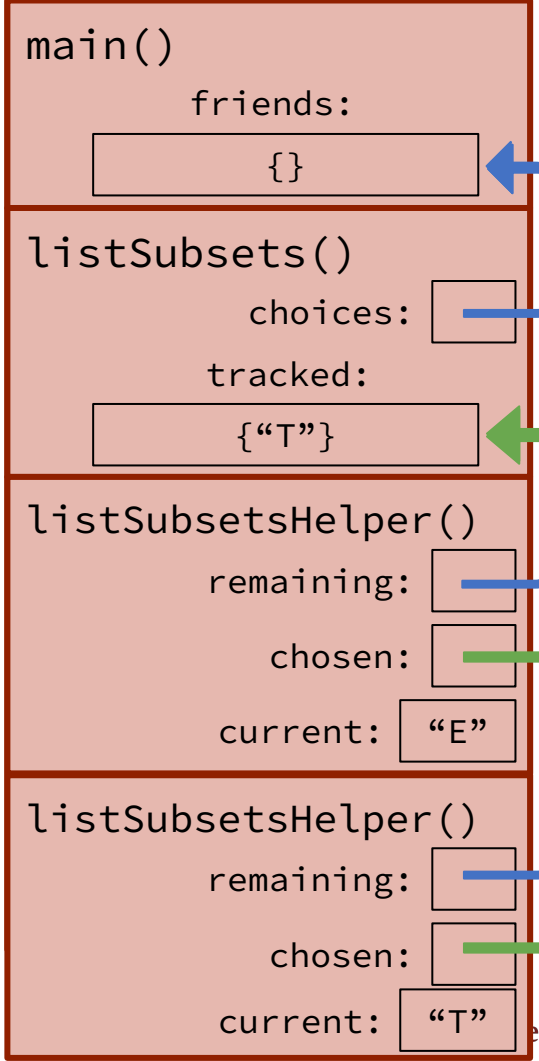
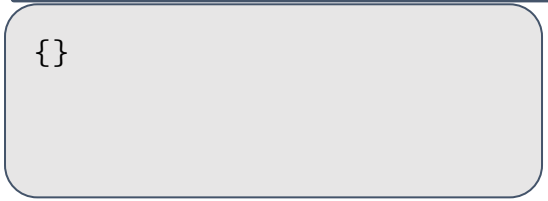
chosen:

current: "T"


```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```
{}
```

main()

friends:

```
{}
```

listSubsets()

choices:

tracked:

```
{"T"}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

```
{}
```

main()

friends:

```
{}
```

listSubsets()

choices:

tracked:

```
{"T"}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

51

iversity

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```
{}
```

```
listSubsetsHelper()
```

```

remaining: 
chosen: 

```

```
main()
```

```
friends:
```

```
{}
```

```
listSubsets()
```

```
choices: 
```

```
tracked:
```

```
{"T"}
```

```
listSubsetsHelper()
```

```
remaining: 
```

```
chosen: 
```

```
current: "E"
```

```
listSubsetsHelper()
```

```
remaining: 
```

```
chosen: 
```

```
current: "T"
```

```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

{}

listSubsetsHelper()

remaining:

chosen:

main()

friends:

{}

listSubsets()

choices:

tracked:

{"T"}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```
{}
```

```
listSubsetsHelper()
```

```

remaining: [ ]
chosen: [ ]

```

```
main()
```

```
friends:
```

```
{}
```

```
listSubsets()
```

```
choices: [ ]
```

```
tracked:
```

```
{"T"}
```

```
listSubsetsHelper()
```

```
remaining: [ ]
```

```
chosen: [ ]
```

```
current: "E"
```

```
listSubsetsHelper()
```

```
remaining: [ ]
```

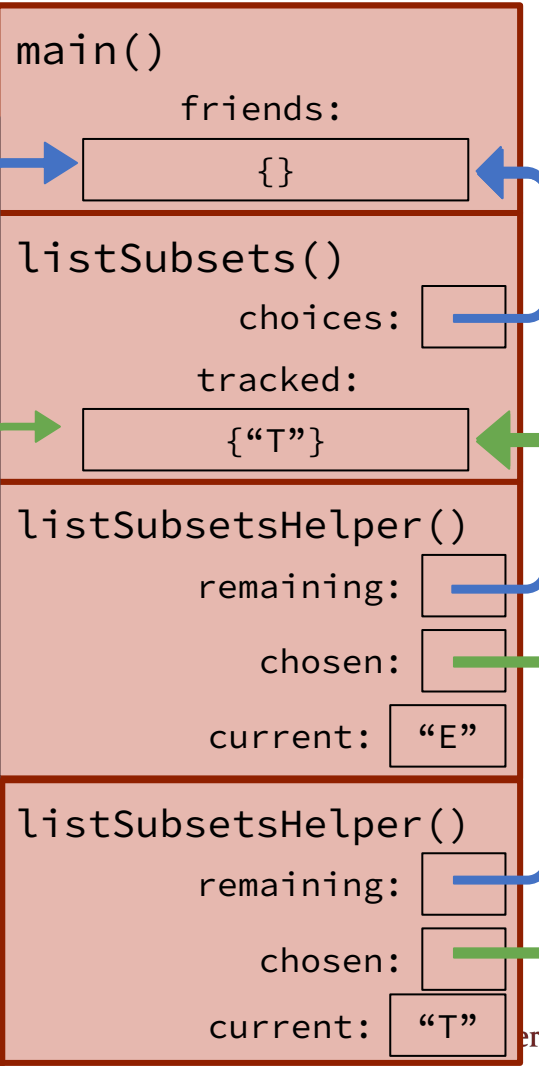
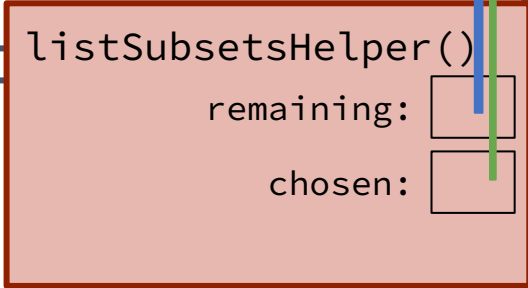
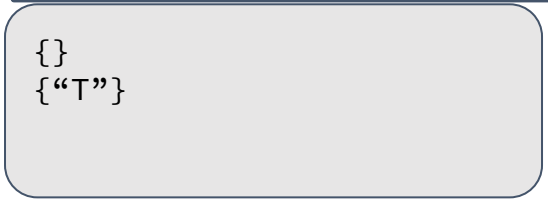
```
chosen: [ ]
```

```
current: "T"
```

```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

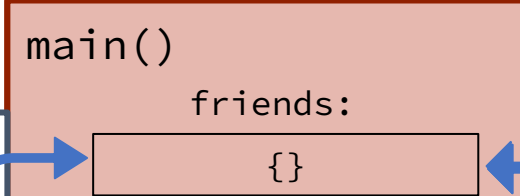
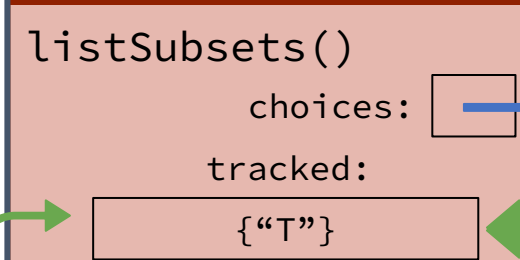
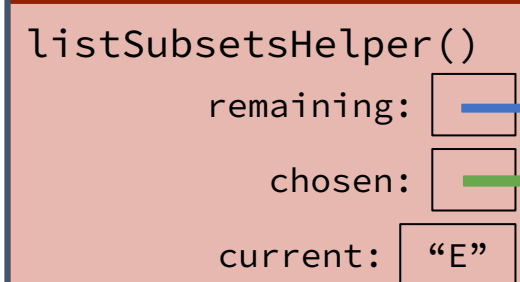
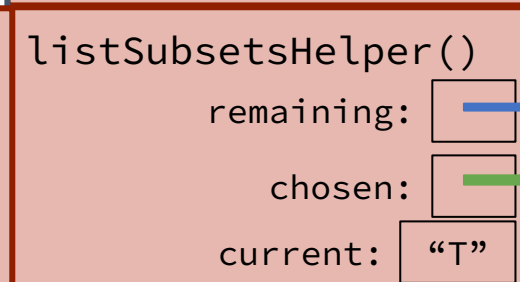
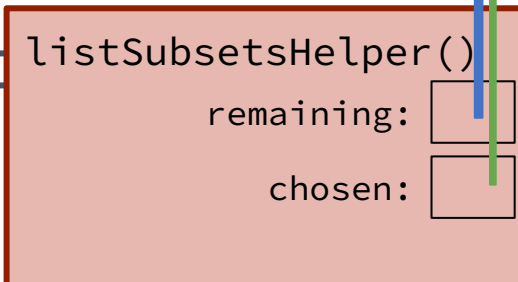
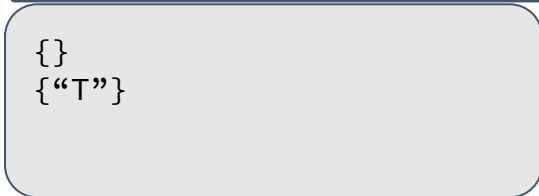
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

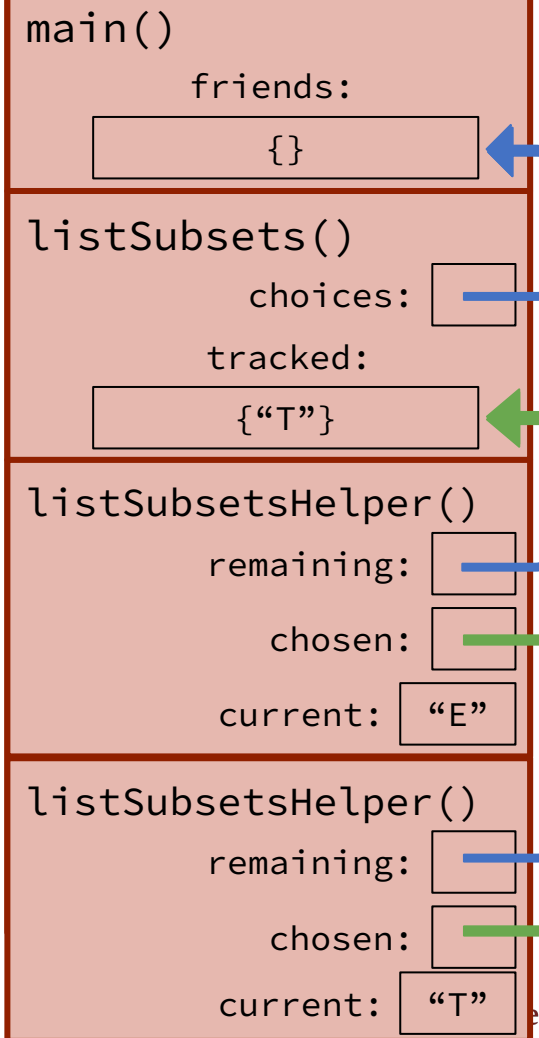
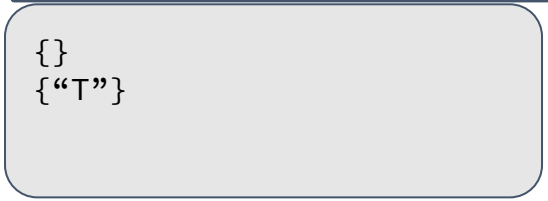
```




```

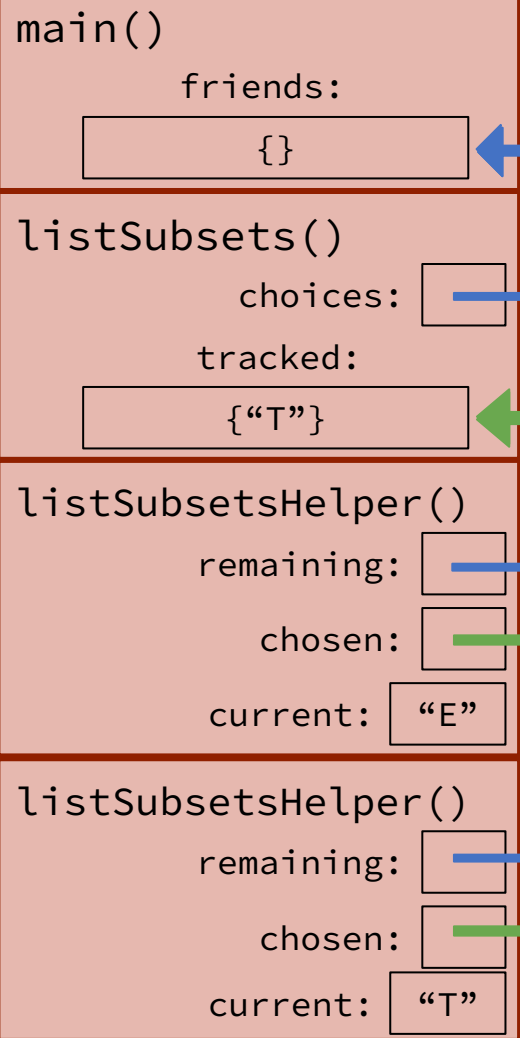
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

{
{"T"}



```

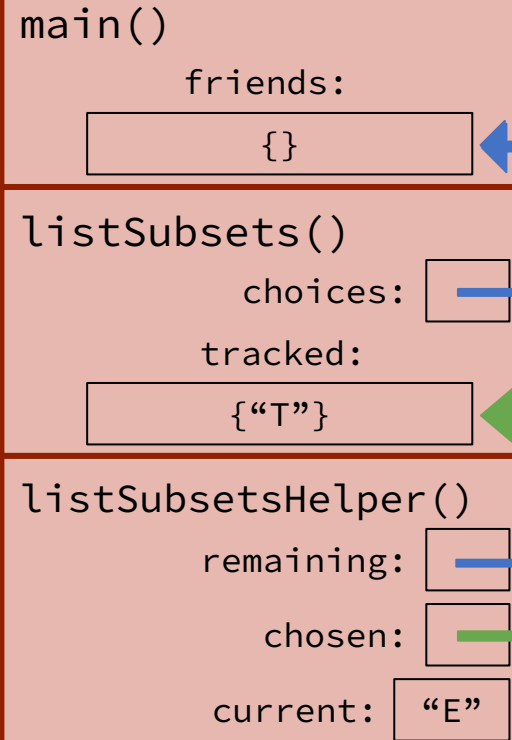
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

```
{ }
{"T"}
```

main()

friends:

```
{ }
```

listSubsets()

choices:

tracked:

```
{"T"}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

60

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}

```

```

main()
    friends:
    {}

listSubsets()
    choices:
    tracked:
    {"E", "T"}

listSubsetsHelper()
    remaining:
    chosen:
    current: "E"

```

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}

```

```

main()
  friends:
  {}

listSubsets()
  choices:
  tracked:
  {"E", "T"}

listSubsetsHelper()
  remaining:
  chosen:
  current: "E"

```

```

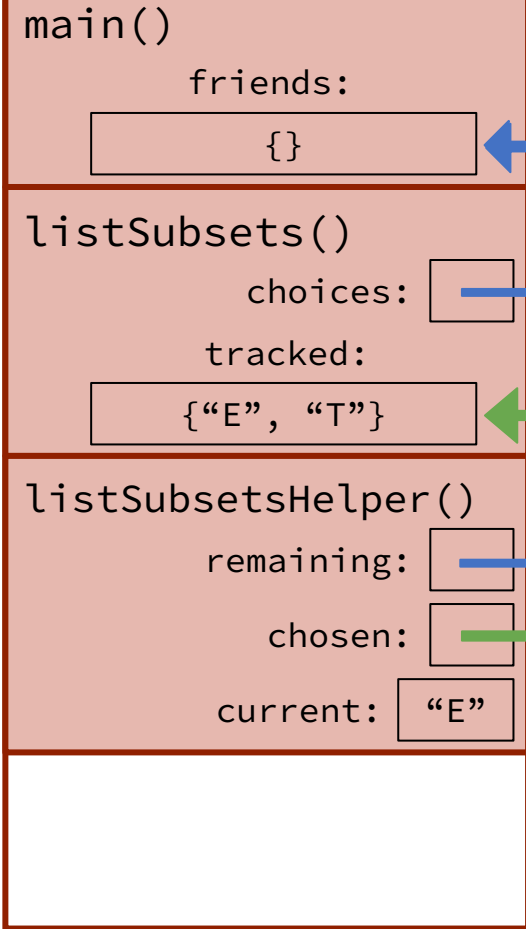
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}
```

```
{ }
{"T"}
```

main()

friends:

```
{ }
```

listSubsets()

choices:

tracked:

```
{"E", "T"}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

64


```

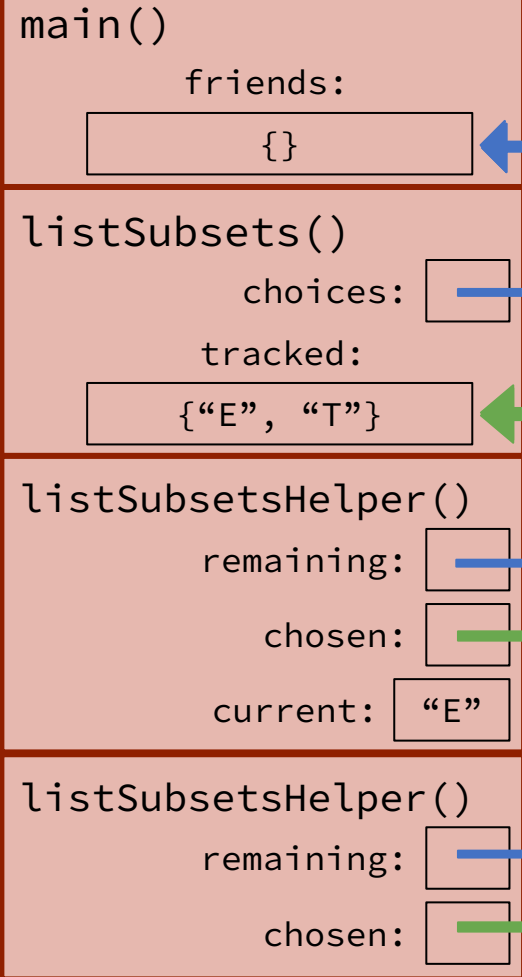
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}

```



```

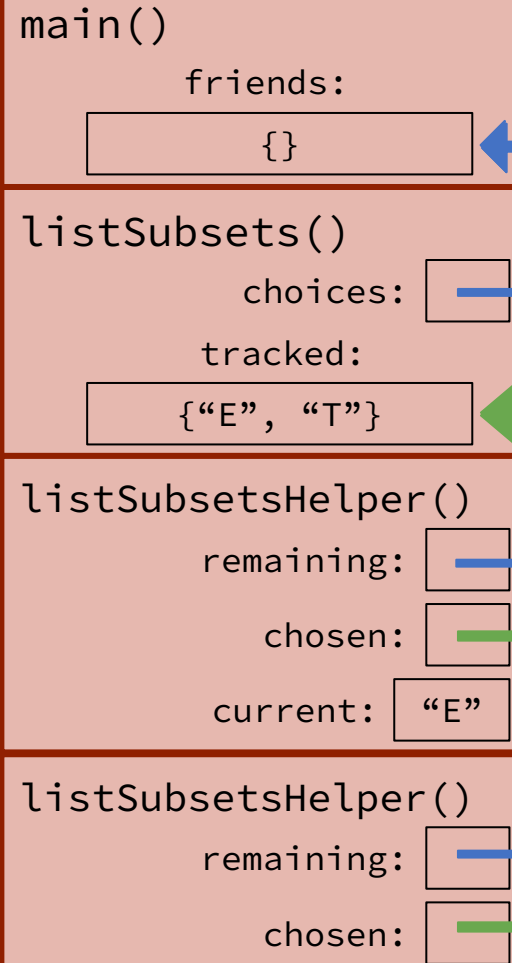
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}

```



```

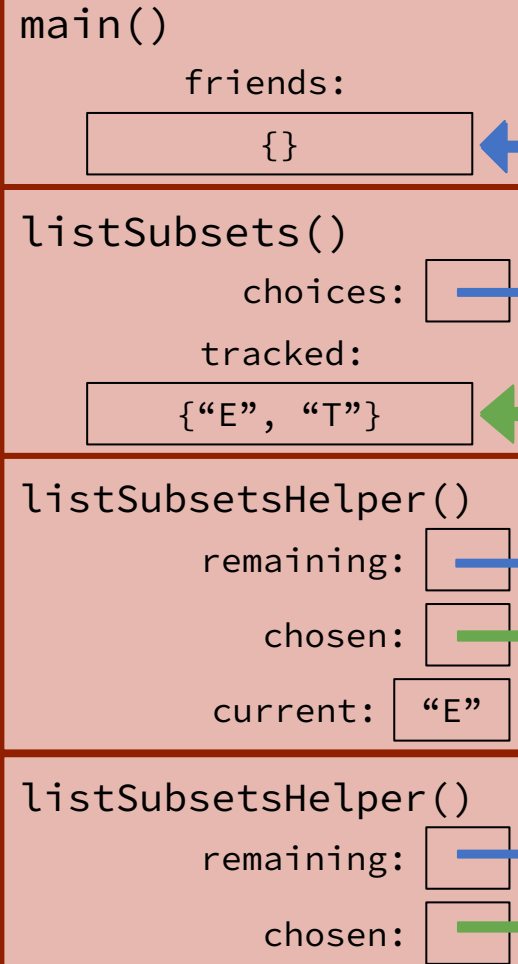
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}
{"E", "T"}

```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}
{"E", "T"}

```

main()

friends:

{}

listSubsets()

choices:

tracked:

{"E", "T"}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

68

```

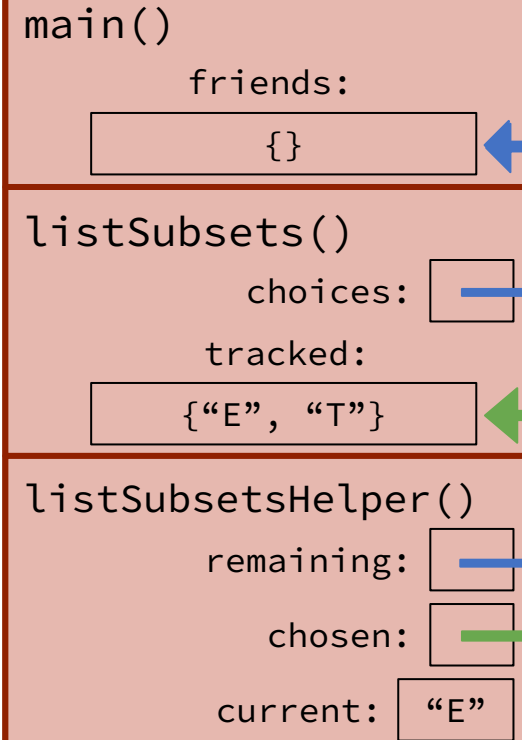
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

```

{}
{"T"}
{"E", "T"}

```

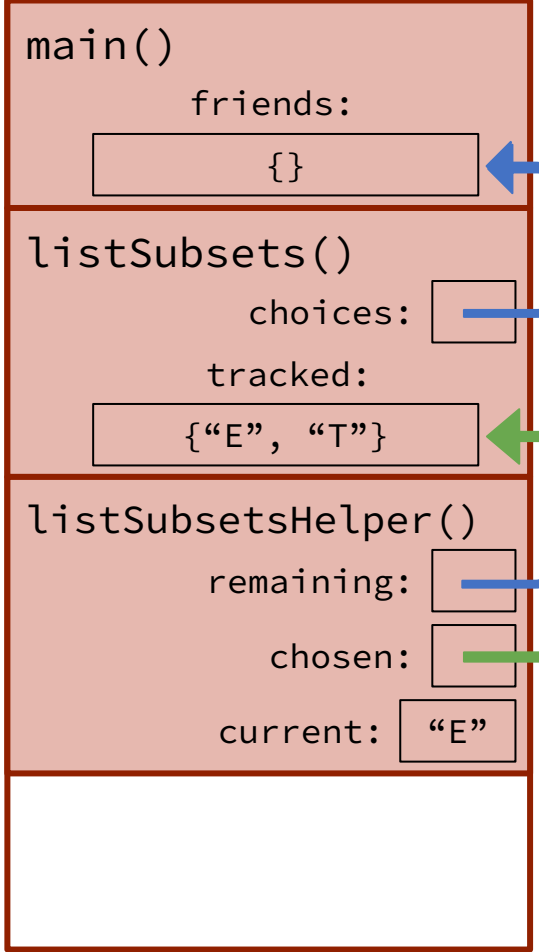


```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
}

```

- {}
- {"T"}
- {"E", "T"}



```

void listSubsets(Set<string>& choices) {
    Set<string> tracked;
    listSubsetsHelper(choices, tracked);
}

```

main()

friends:

{}

listSubsets()

choices:

tracked:

{"E", "T"}

```

{}
{"T"}
{"E", "T"}

```

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{}

listSubsets()

choices:

tracked:

{"E", "T"}

```
{  
{ "T" }  
{ "E", "T" }
```



```
int main () {  
    Set<string> friends = {'A', 'E', 'T'};  
    listSubsets(friends);  
    return 0;  
}
```

```
{}  
{"T"}  
{"E", "T"}
```

main()

friends:

```
{}
```

```
int main () {  
    Set<string> friends = {'A', 'E', 'T'};  
    listSubsets(friends);  
    return 0;  
}
```

```
{}  
{"T"}  
{"E", "T"}
```

main()

friends:

```
{}
```

```
{}  
{"T"}  
{"E", "T"}
```

Tracing through Fixed Code!

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

```
main()
```

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()


```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()

friends:

{"E", "T"}

```
int main () {  
    Set<string> friends = {"E", "T"};  
    listSubsets(friends);  
    return 0;  
}
```

main()

friends:

{"E", "T"}

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

```
{“E”, “T”}
```

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

```
void listSubsetsHelper(Set<string>& remaining,  
                      Set<string>& chosen) {  
    if (remaining.isEmpty()) {  
        cout << chosen << endl;  
        return;  
    }  
    string current = remaining.first();  
    remaining = remaining - current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen + current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen - current;  
    remaining = remaining + current;  
}
```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

main()

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

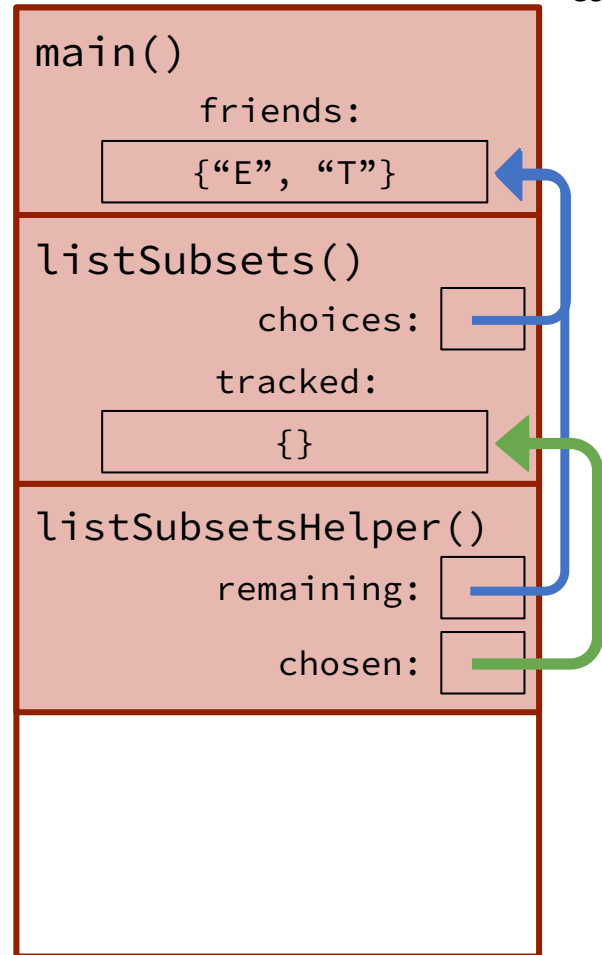
remaining:

chosen:

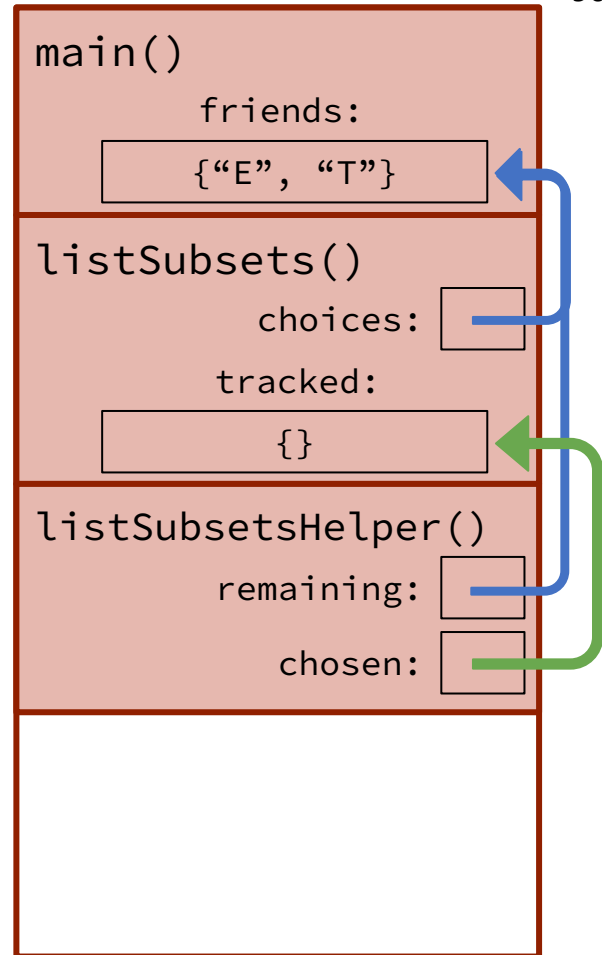

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```



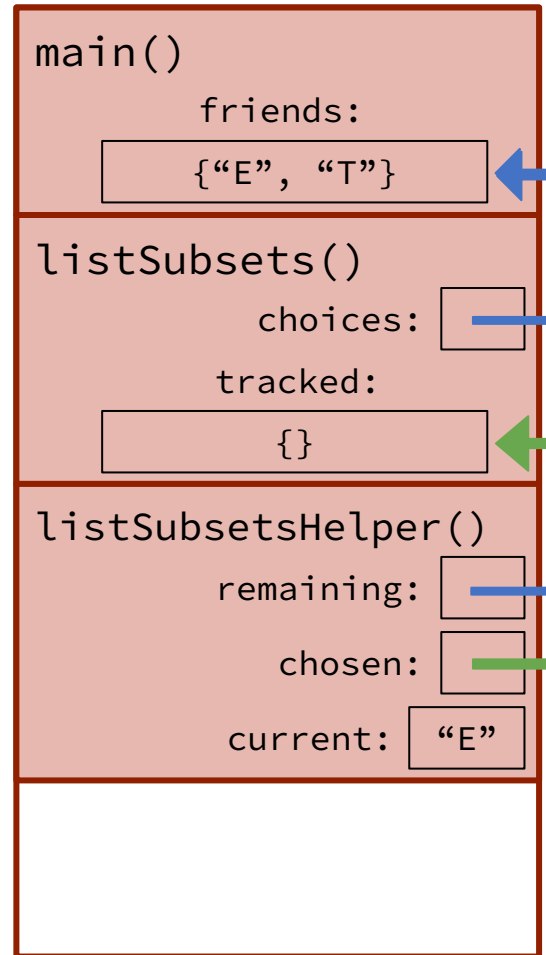
```
void listSubsetsHelper(Set<string>& remaining,  
                      Set<string>& chosen) {  
    if (remaining.isEmpty()) {  
        cout << chosen << endl;  
        return;  
    }  
    string current = remaining.first();  
    remaining = remaining - current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen + current;  
    listSubsetsHelper(remaining, chosen);  
    chosen = chosen - current;  
    remaining = remaining + current;  
}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

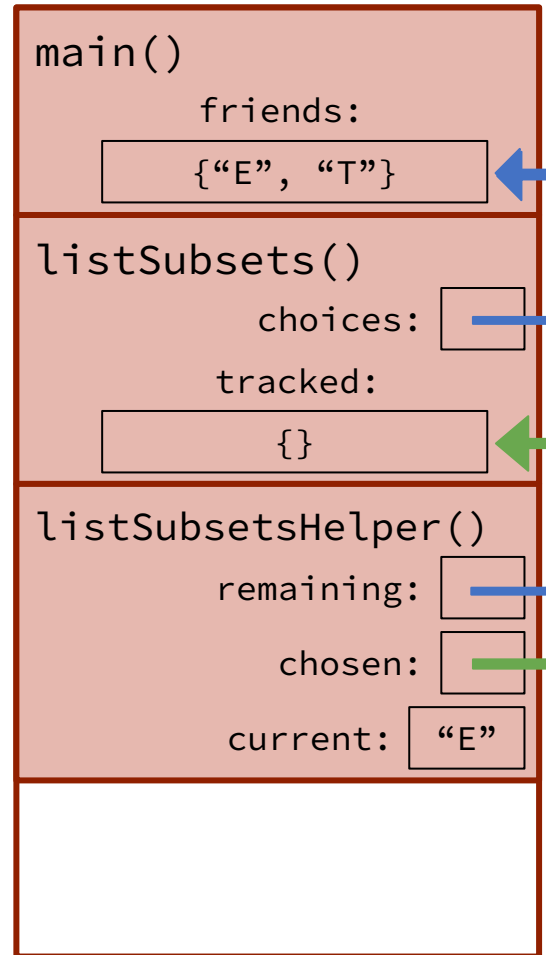
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

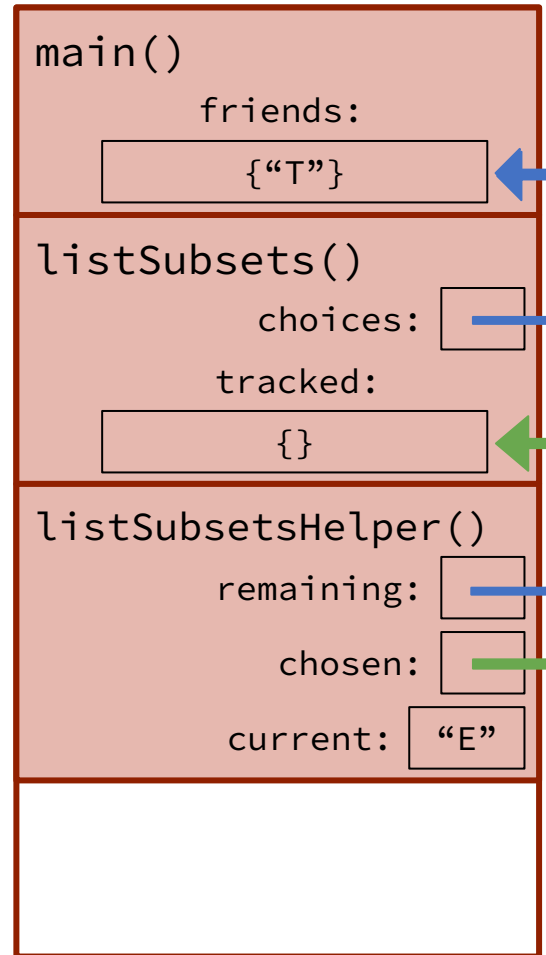
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

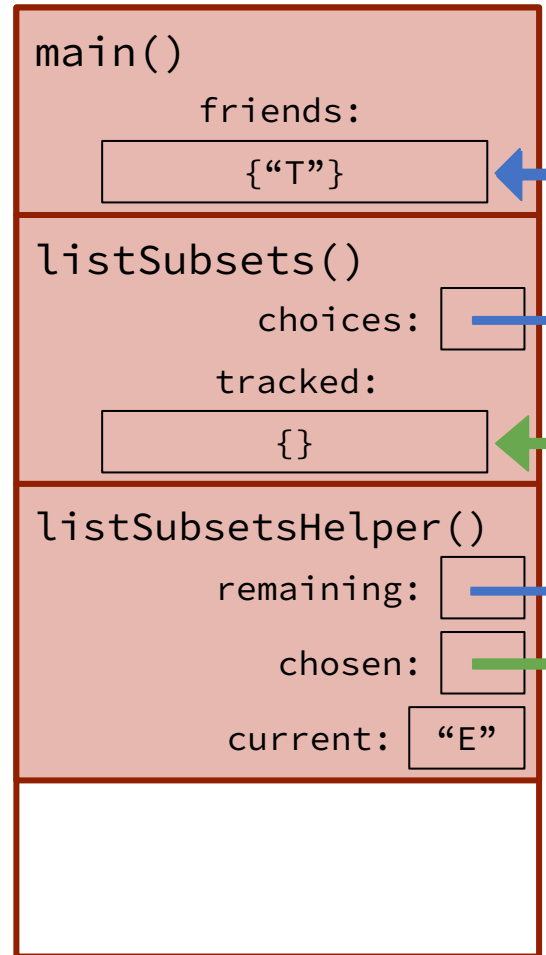
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

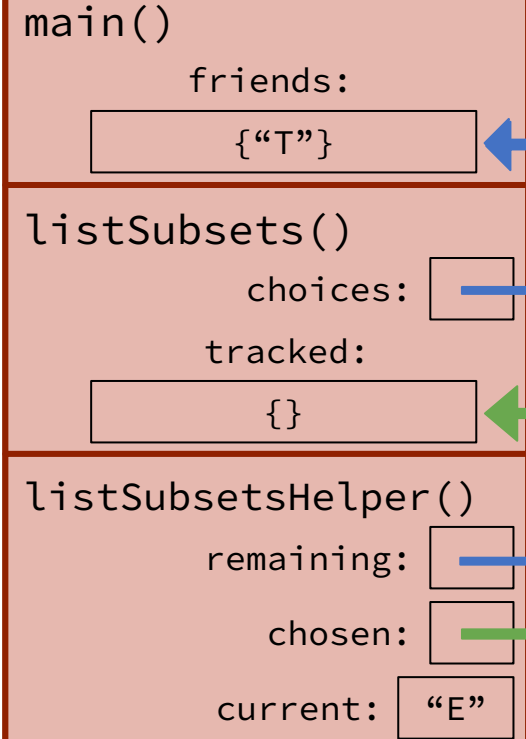
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

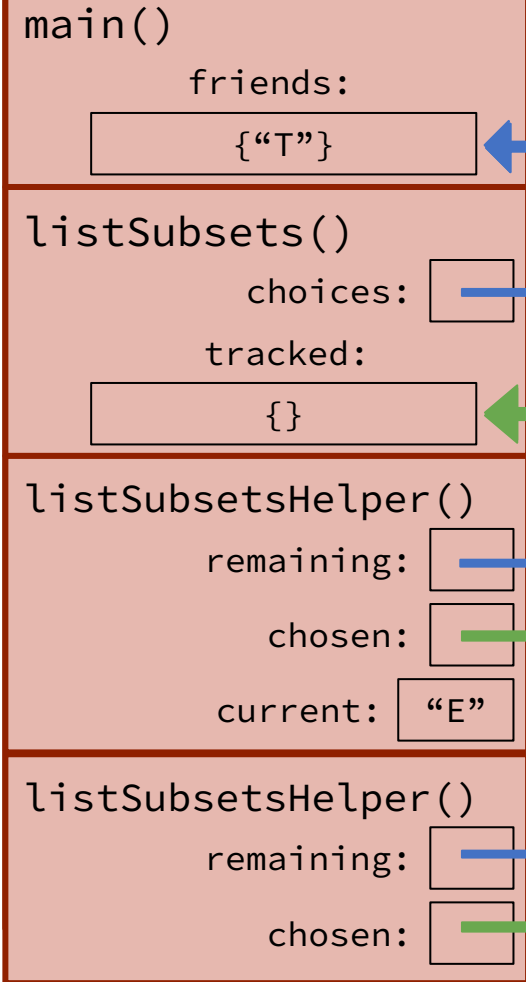
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

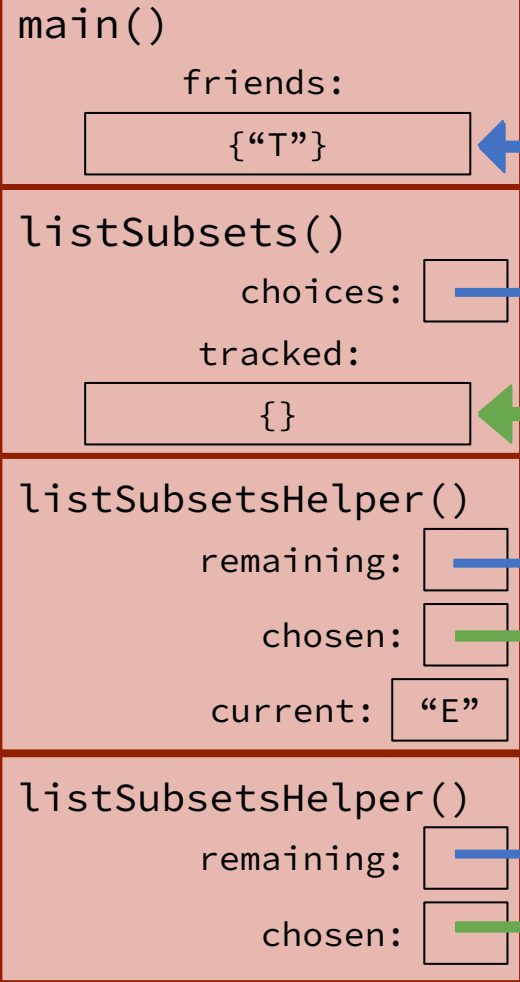
```



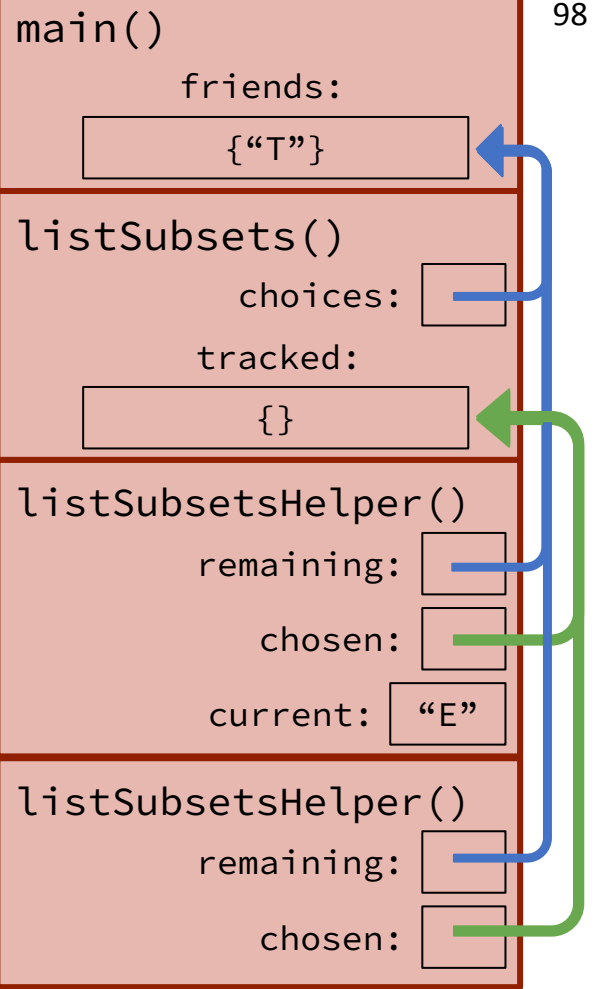

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

main()

friends:

{ "T" }

listSubsets()

choices:

tracked:

{ }

listSubsetsHelper()

remaining:

chosen:

current: "E"

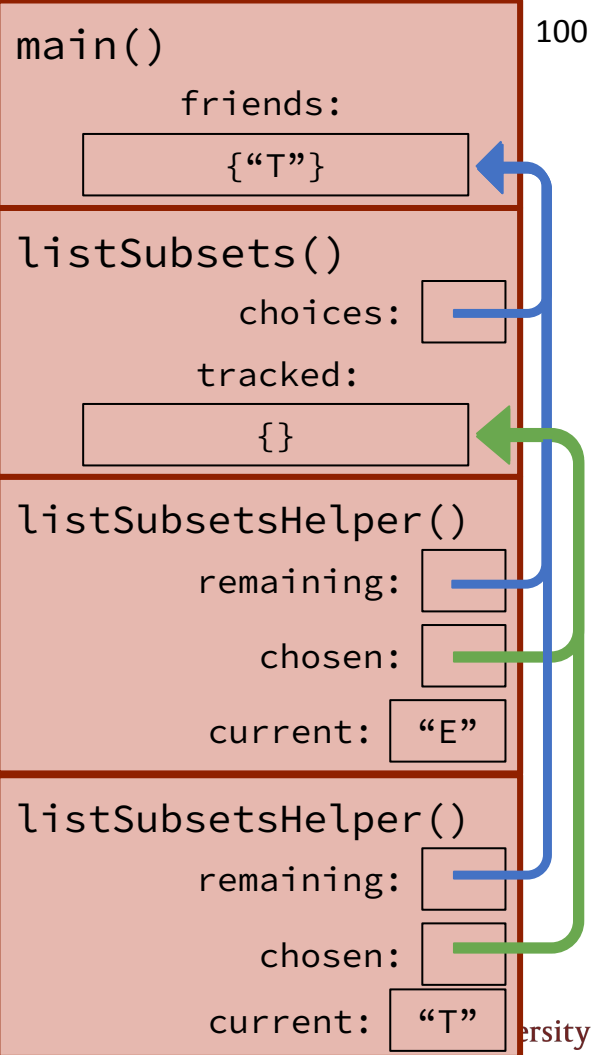
listSubsetsHelper()

remaining:

chosen:

current: "T"

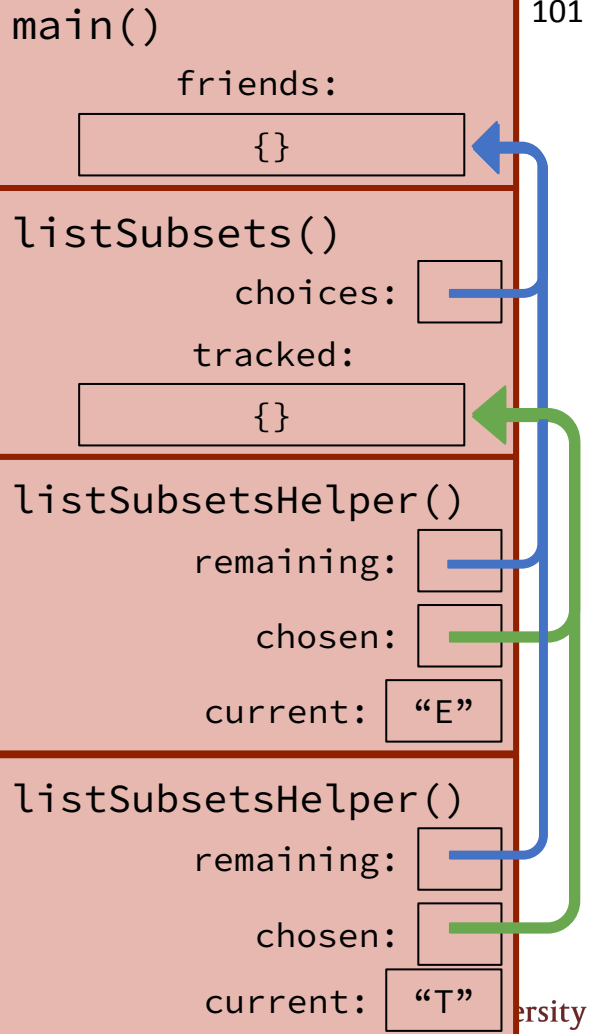
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



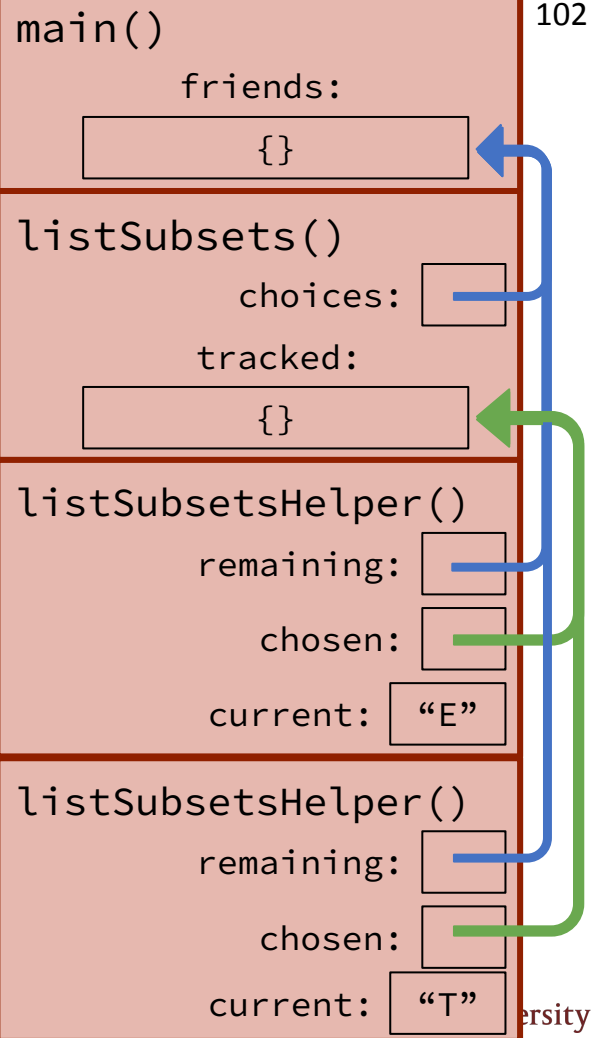
```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```



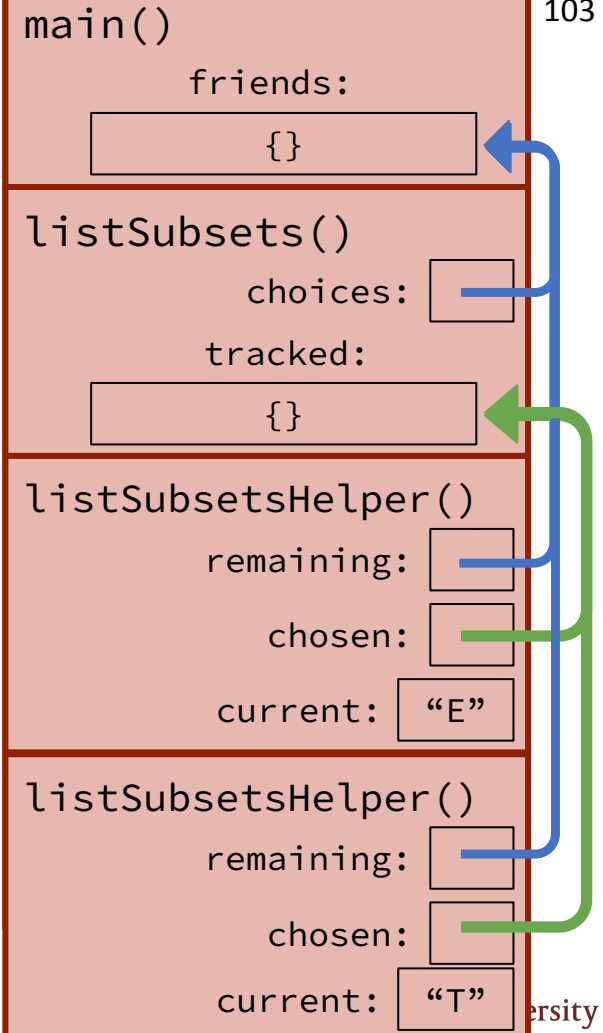
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



```

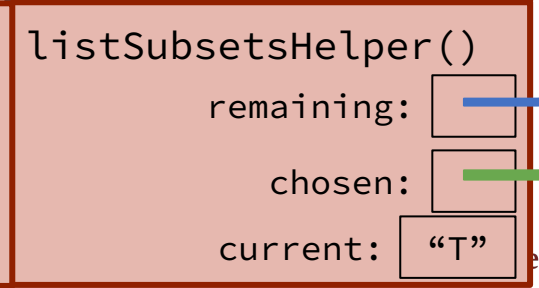
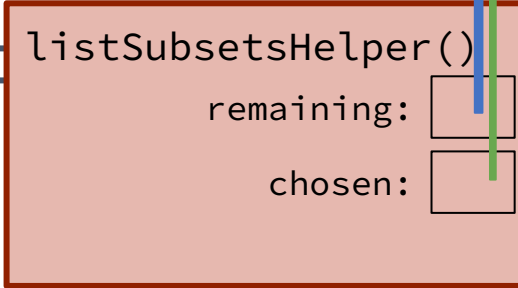
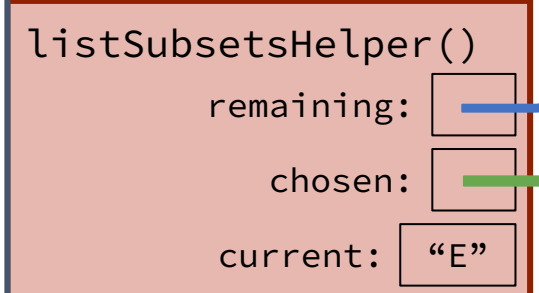
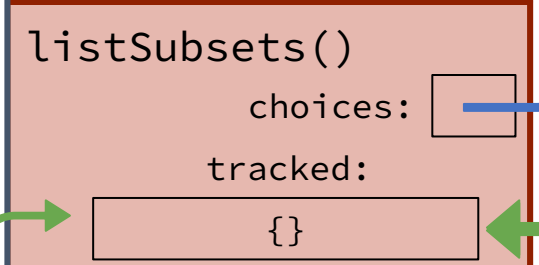
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```



```
void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
```

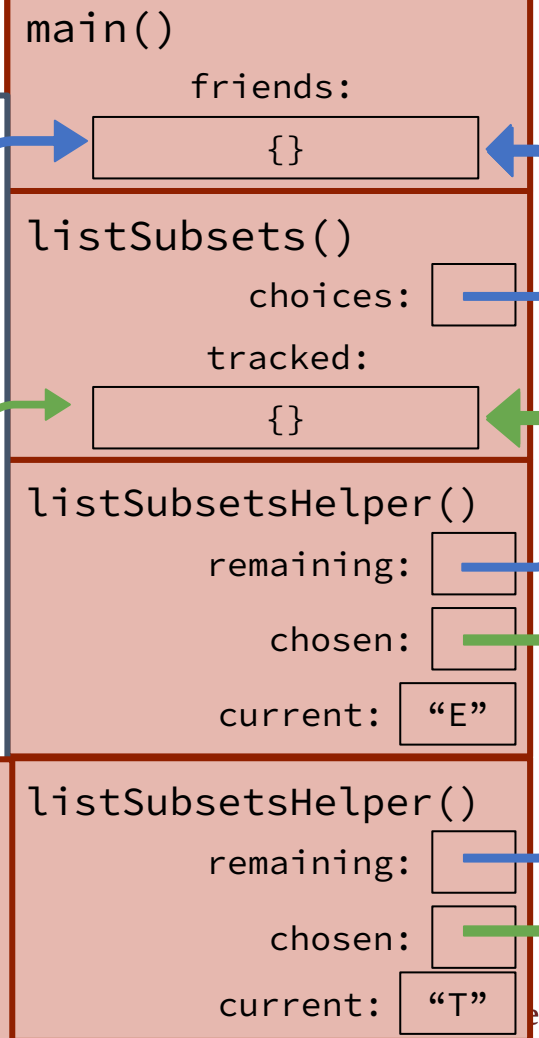
```
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```




```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

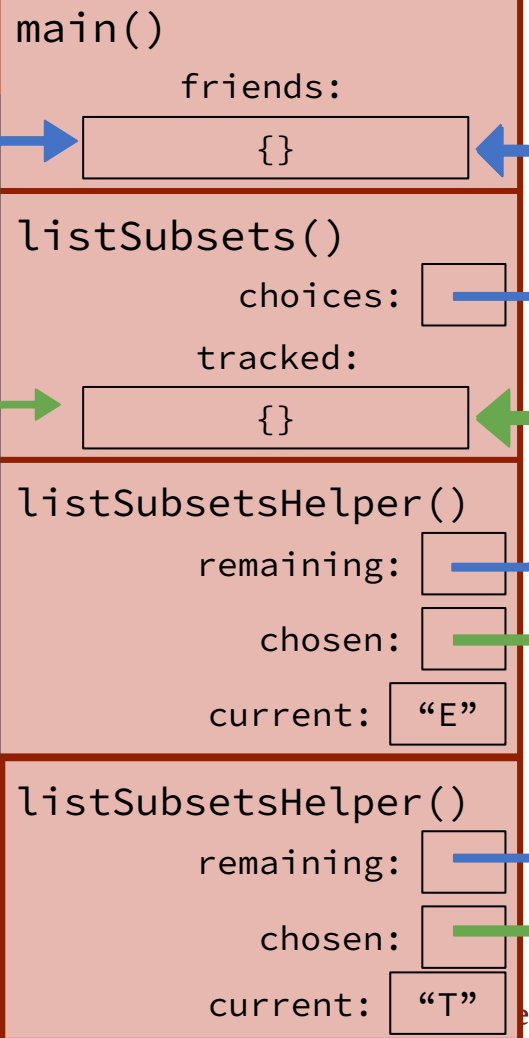
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

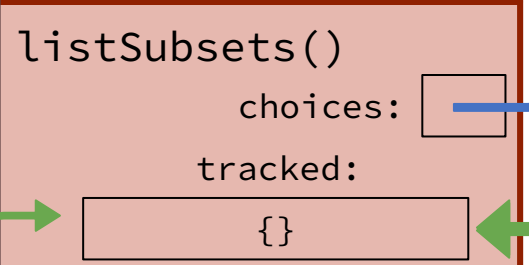
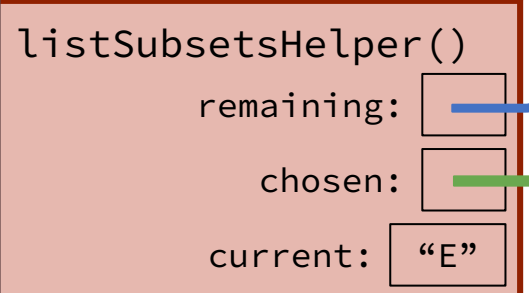
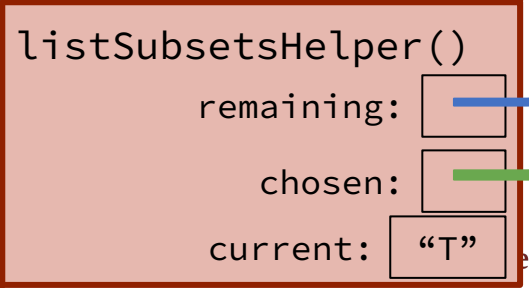
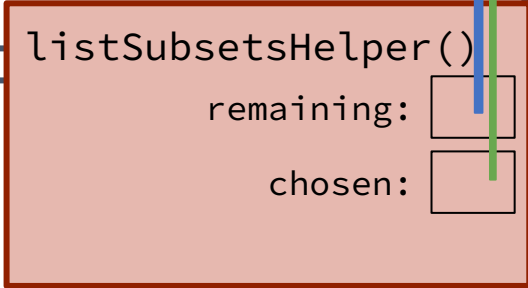
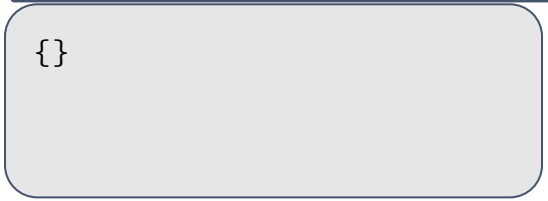
```



```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

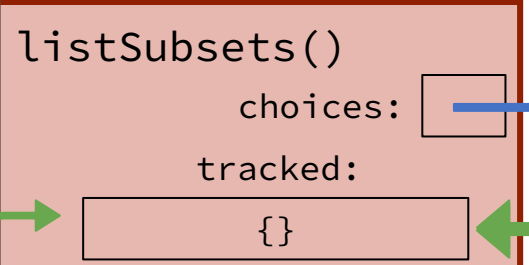
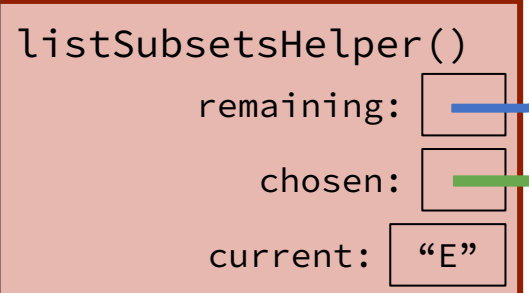
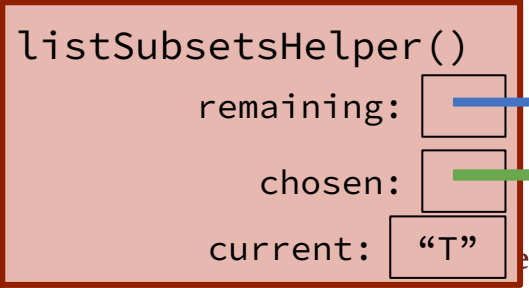
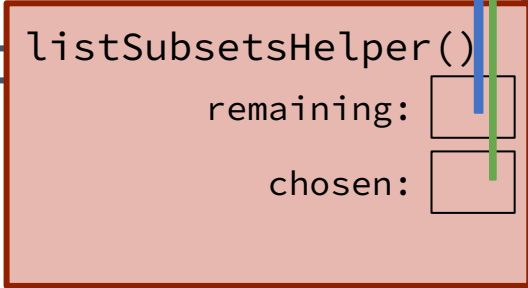
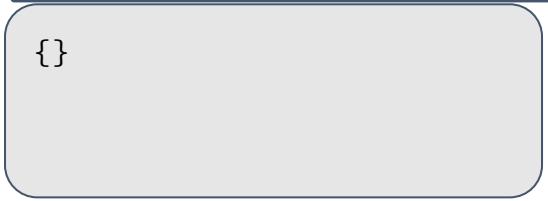
```



```

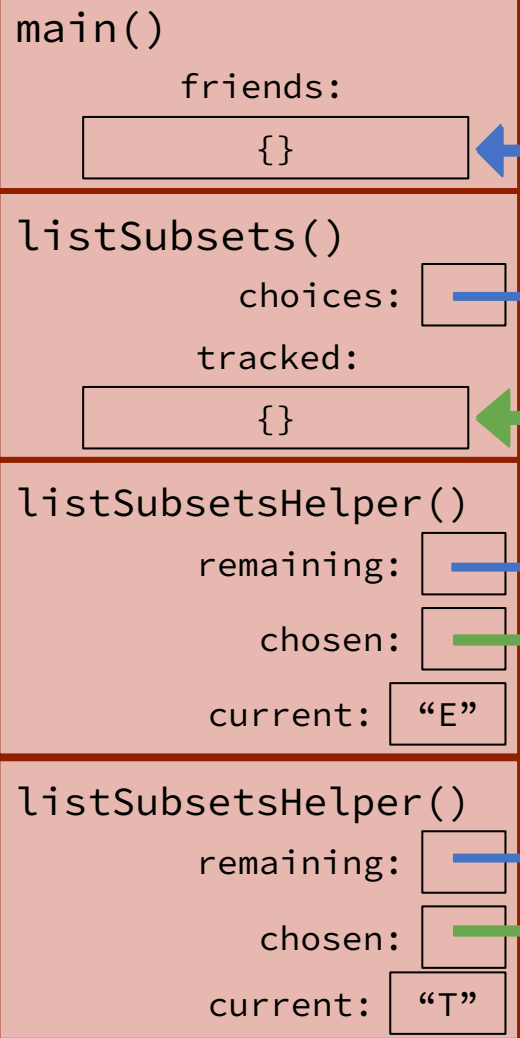
void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```



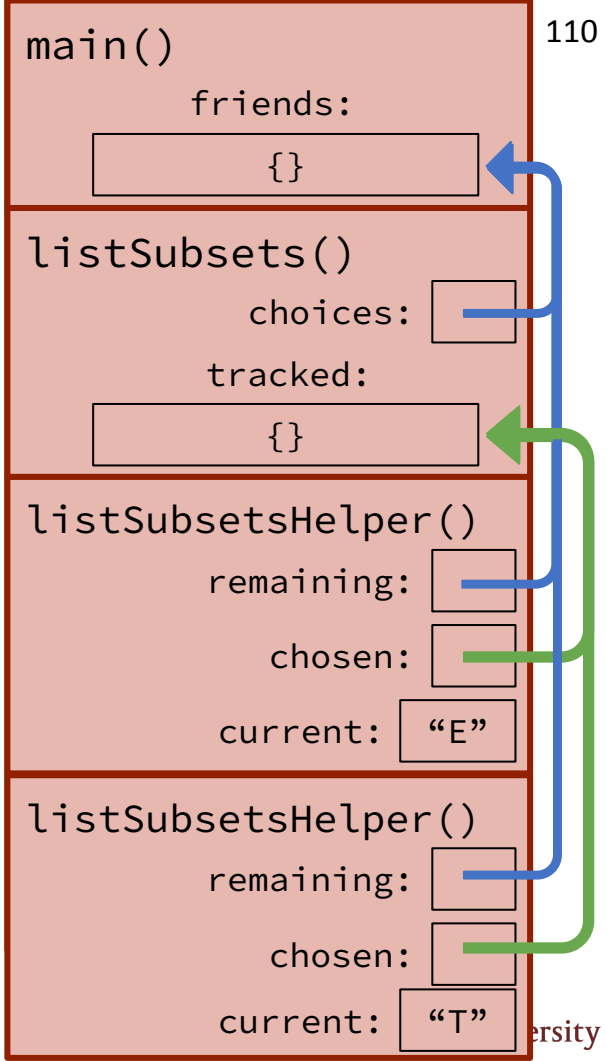
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{}
```



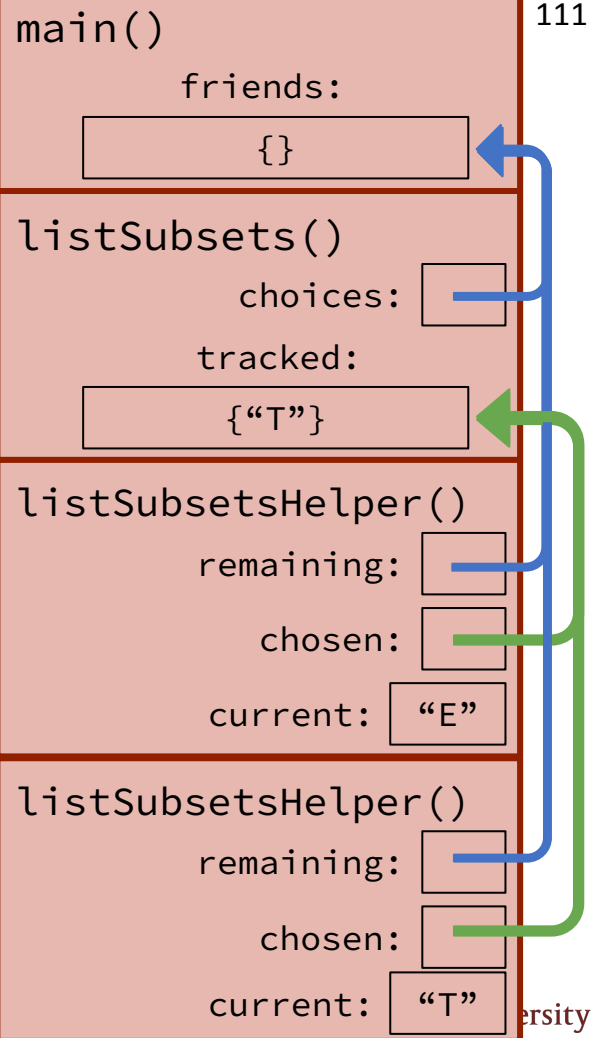
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{}
```



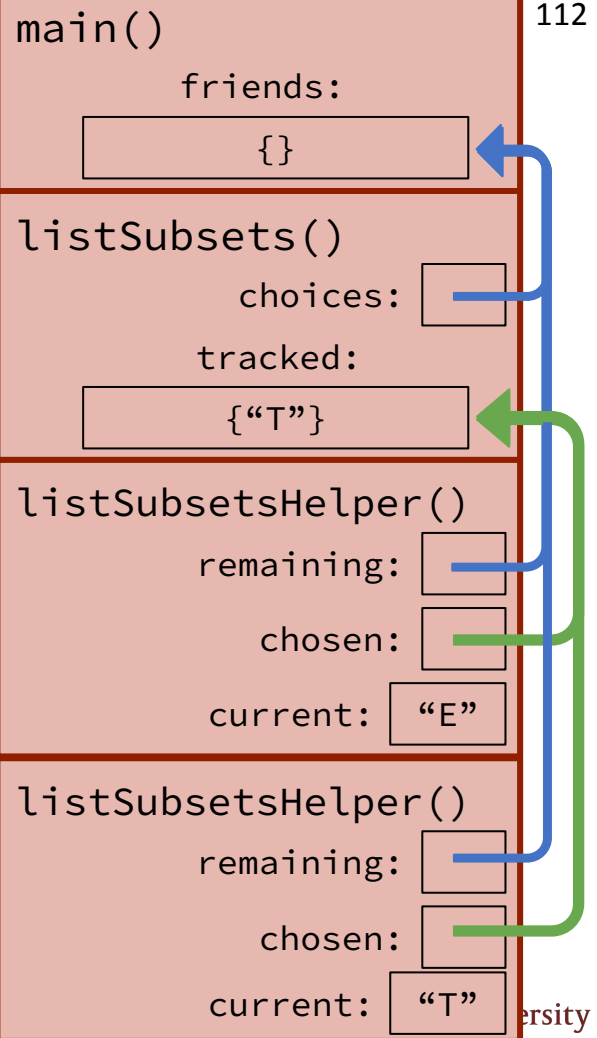
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{}
```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{}
```




```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

{}

main()

113

friends:

{}

listSubsets()

choices:

tracked:

{"T"}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

iversity

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```
{}
```

main() 114

friends:

```
{}
```

listSubsets()

choices:

tracked:

```
{"T"}
```

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

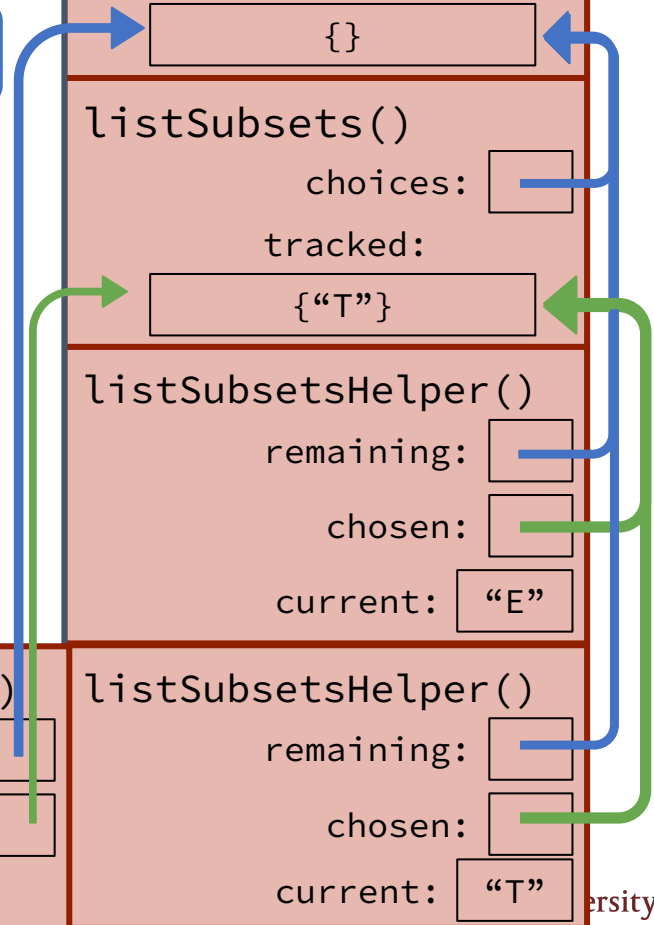
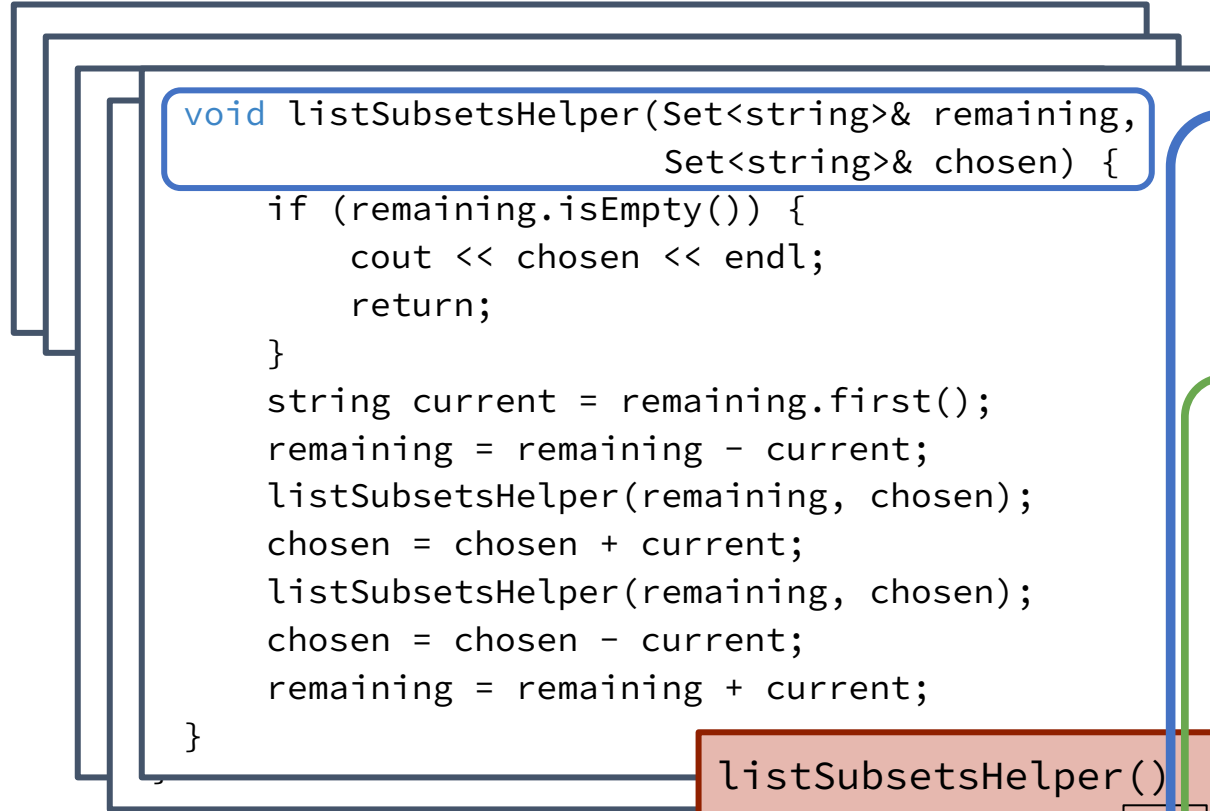
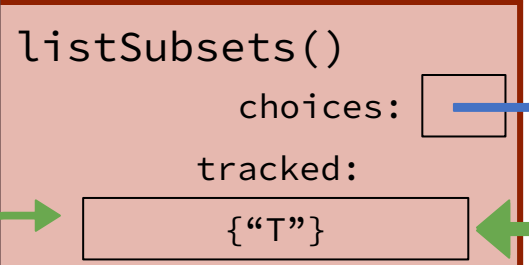
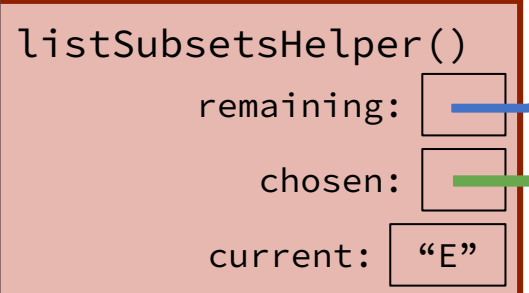
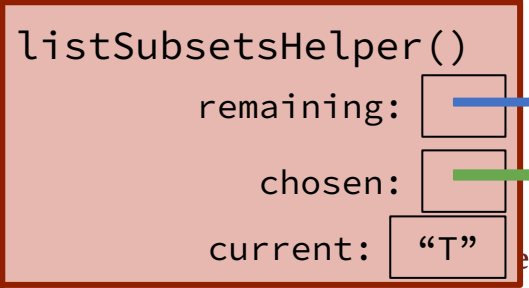
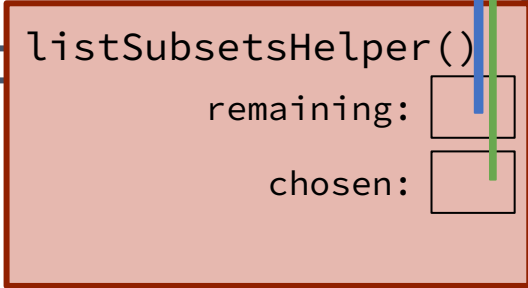
remaining:

chosen:

current: "T" university

```
void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
```

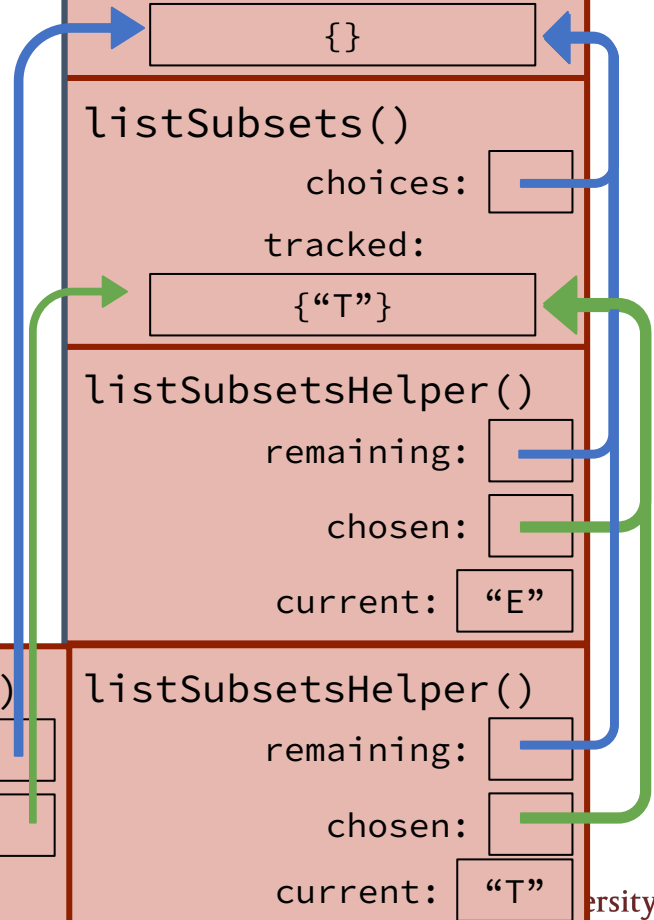
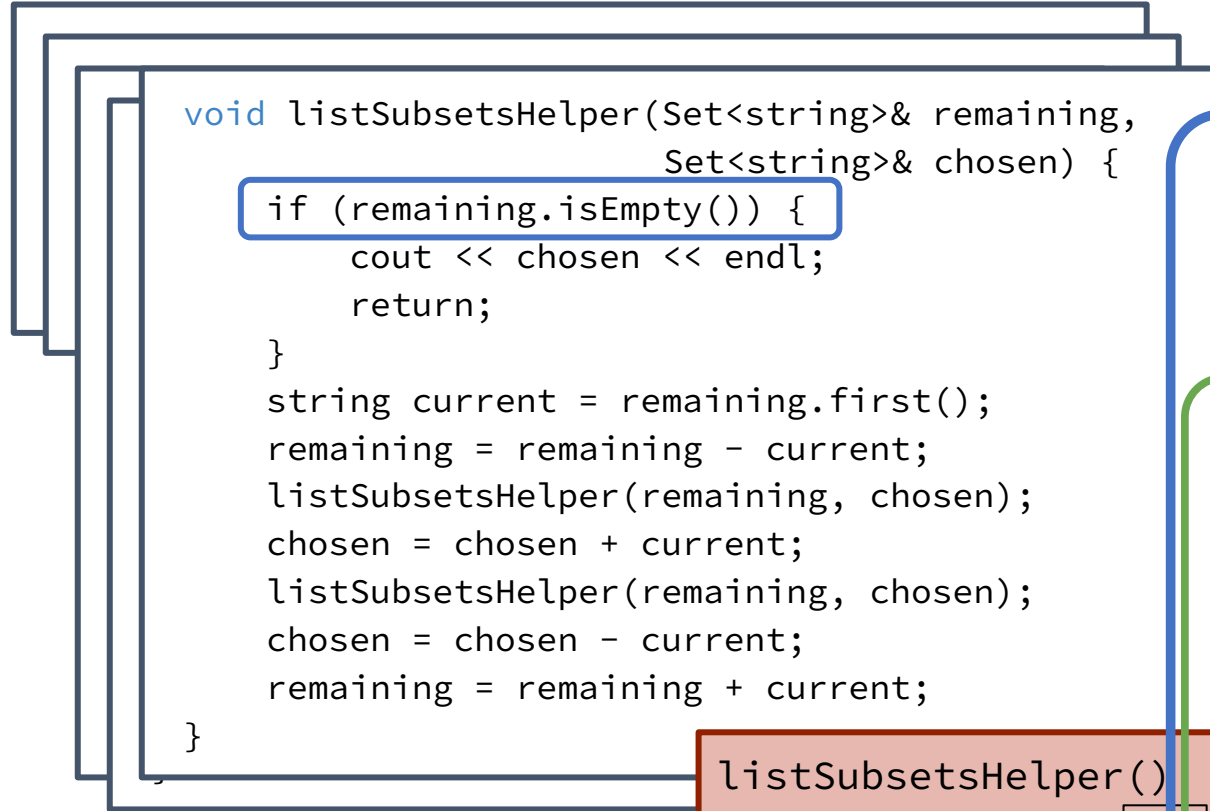
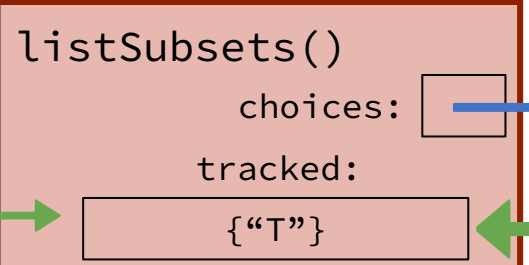
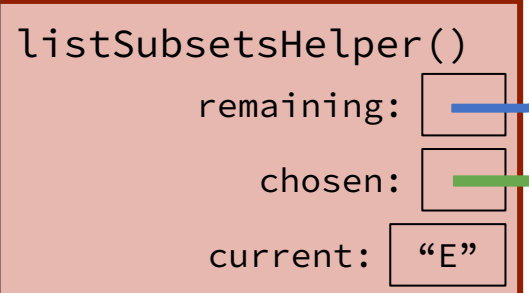
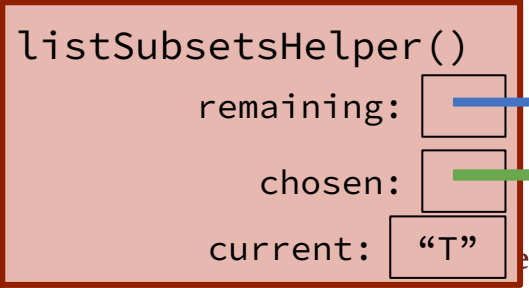
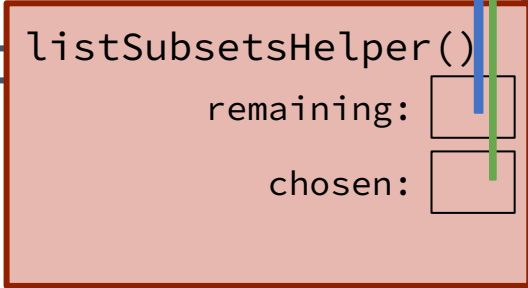
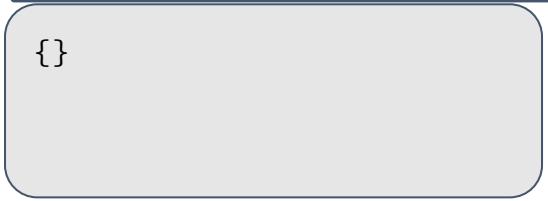
```
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

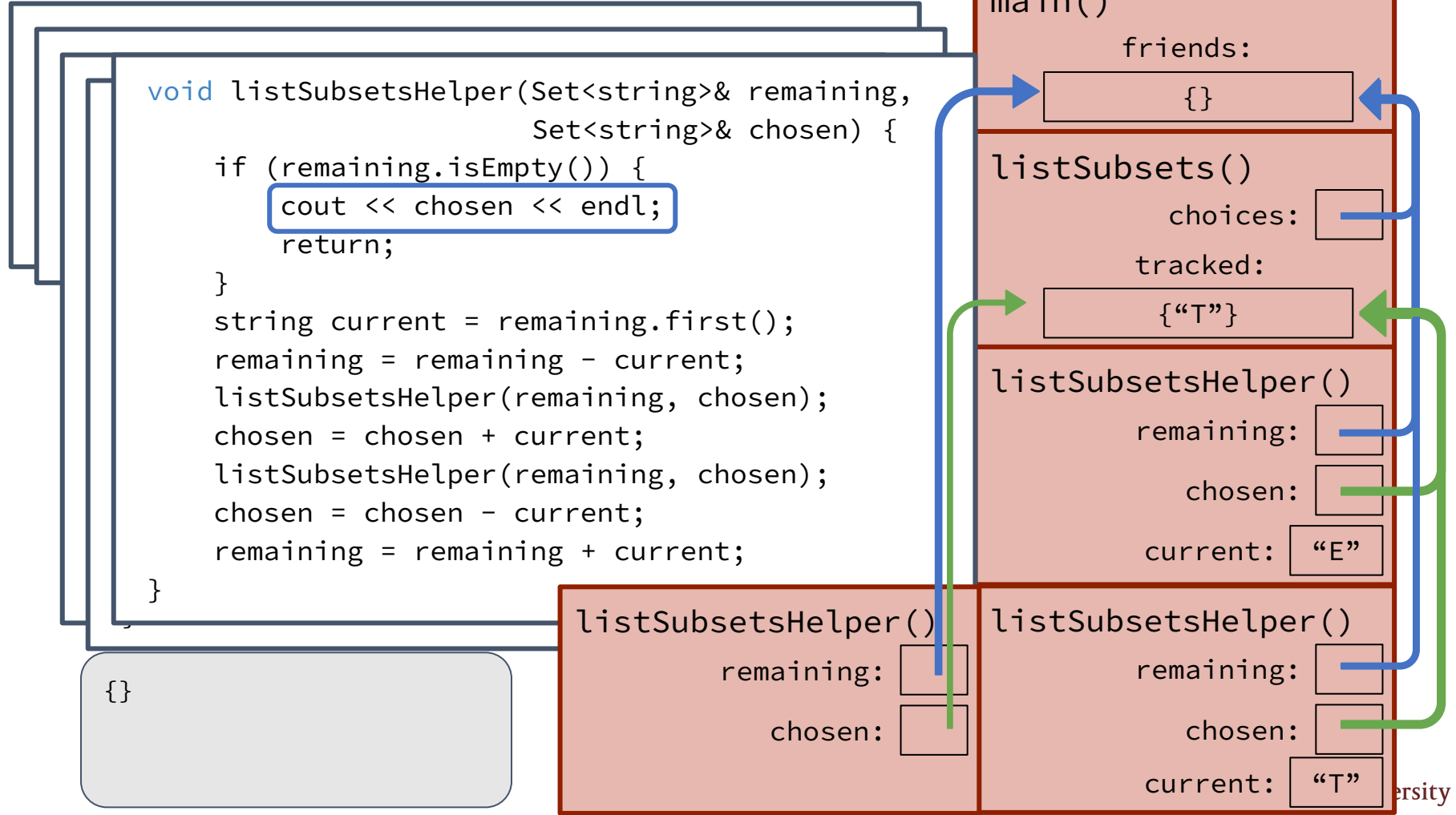
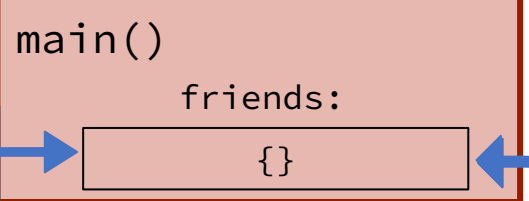
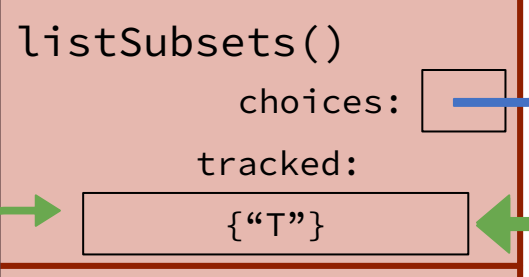
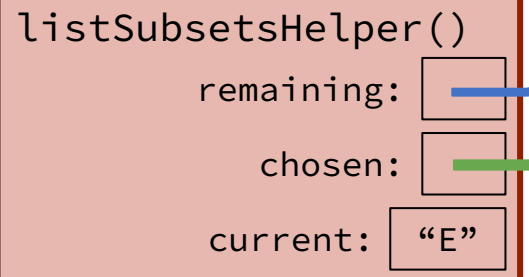
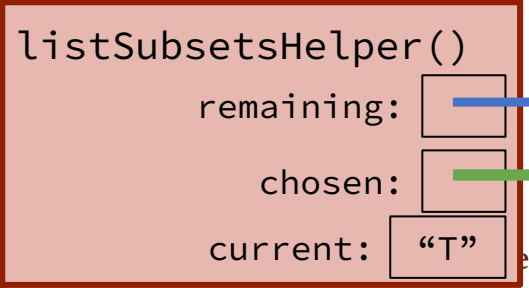
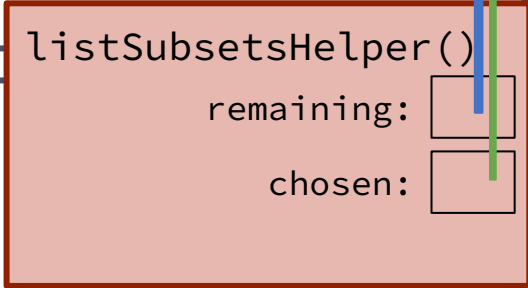
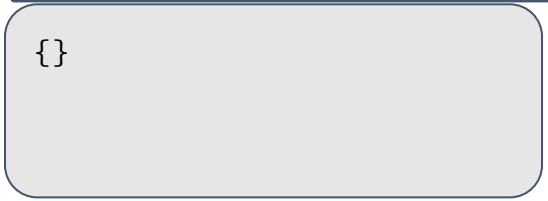
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

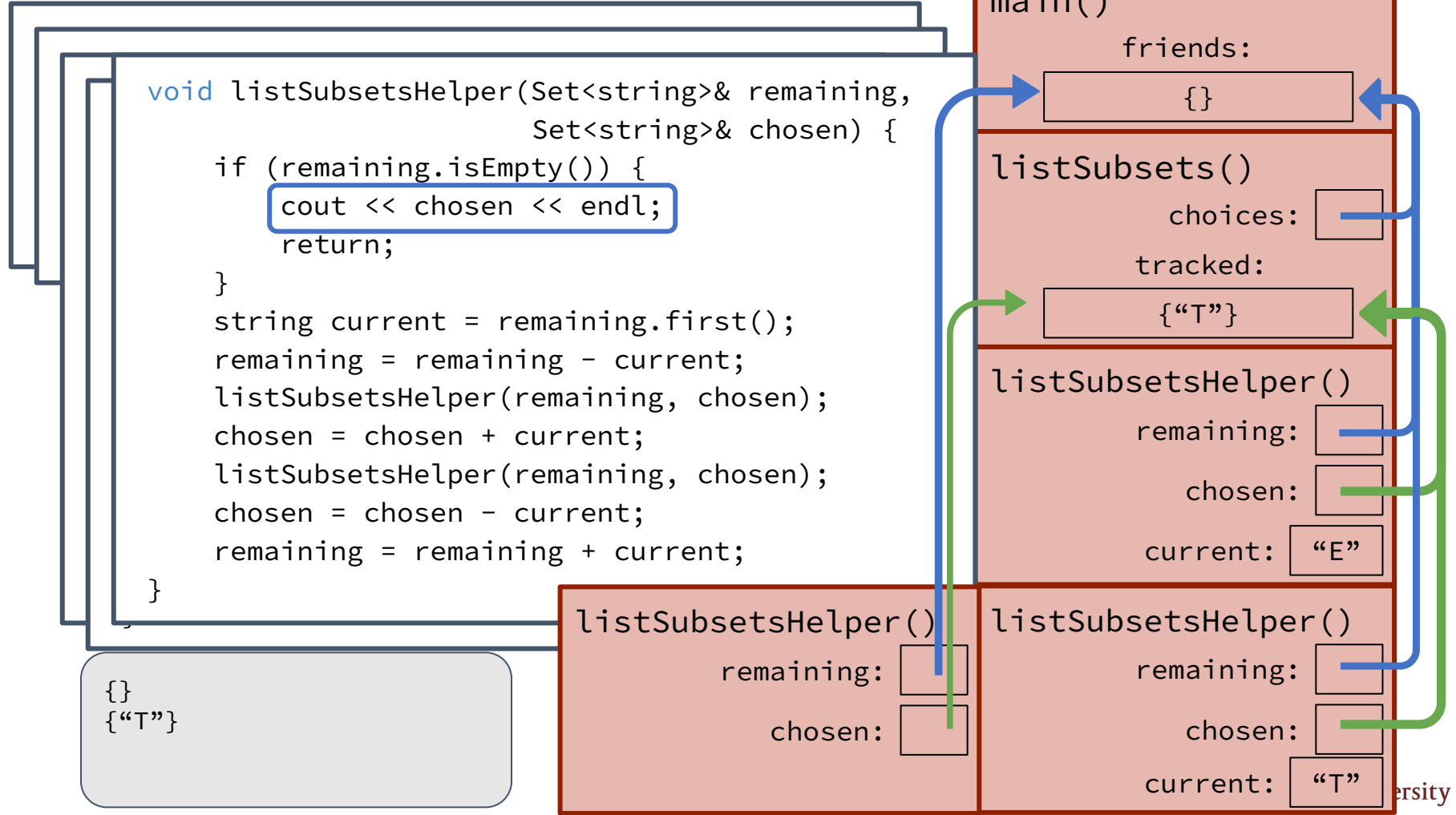
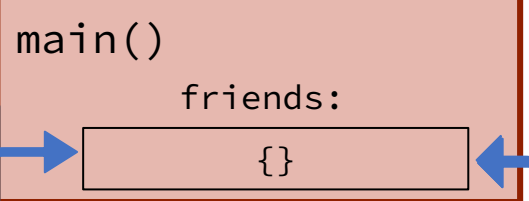
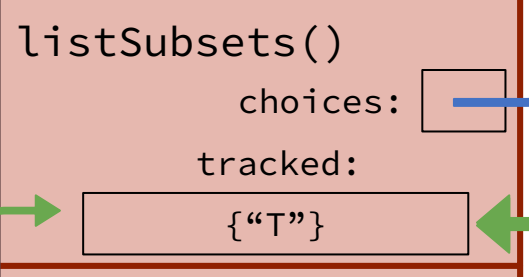
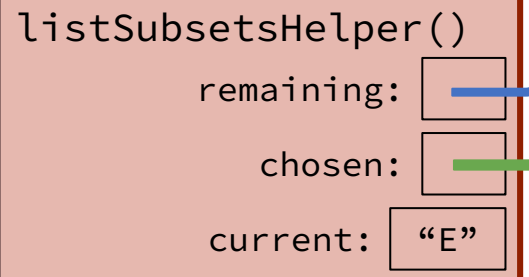
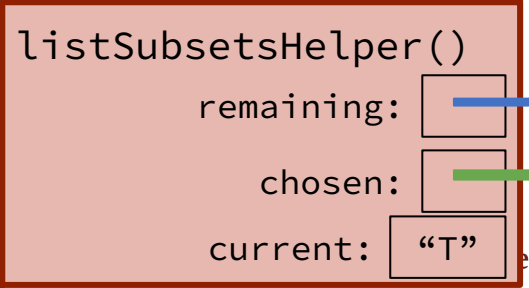
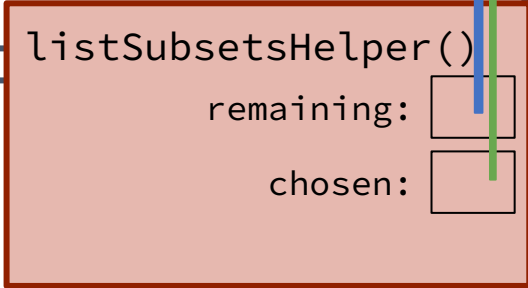
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

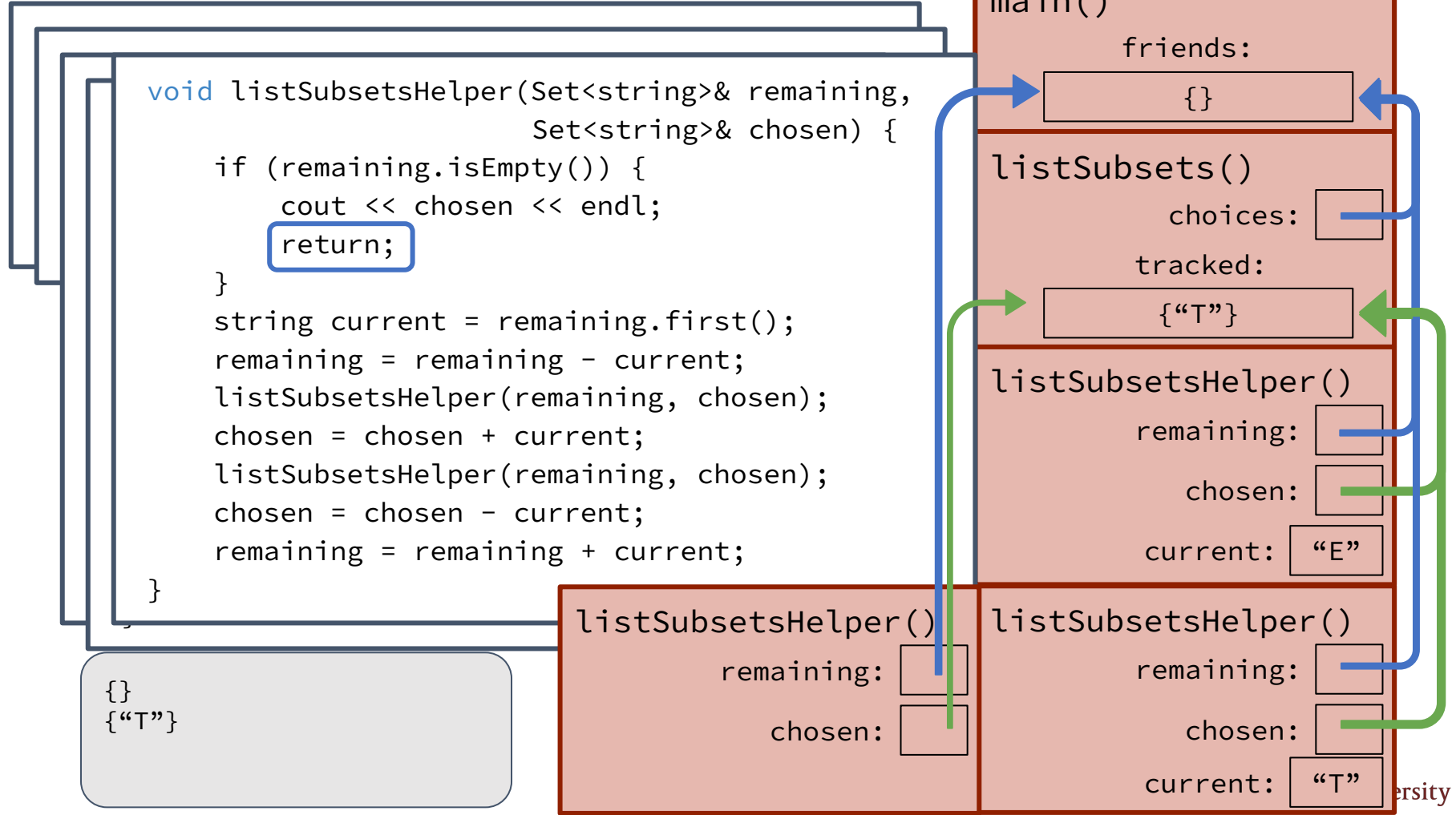
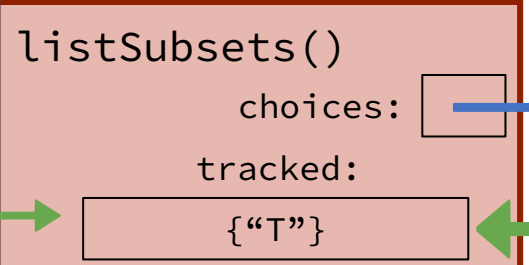
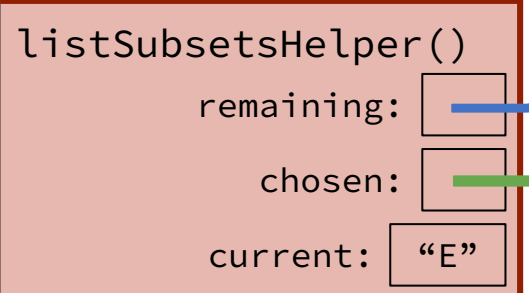
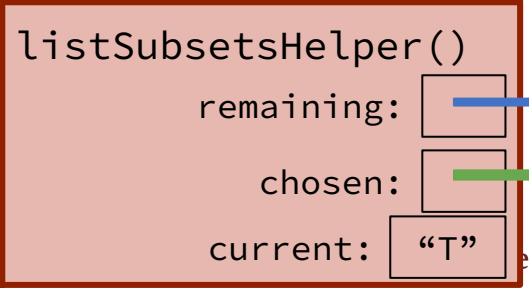
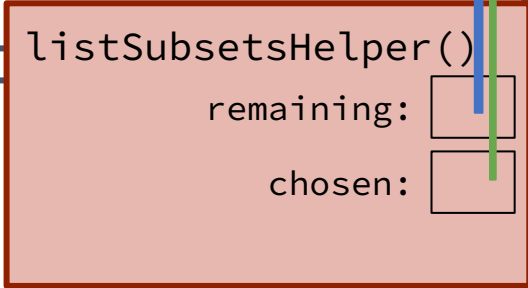
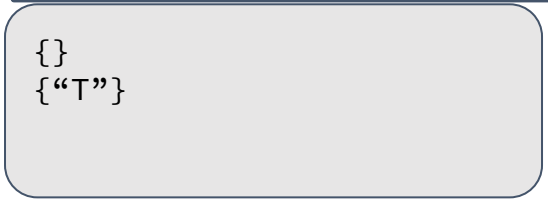
```



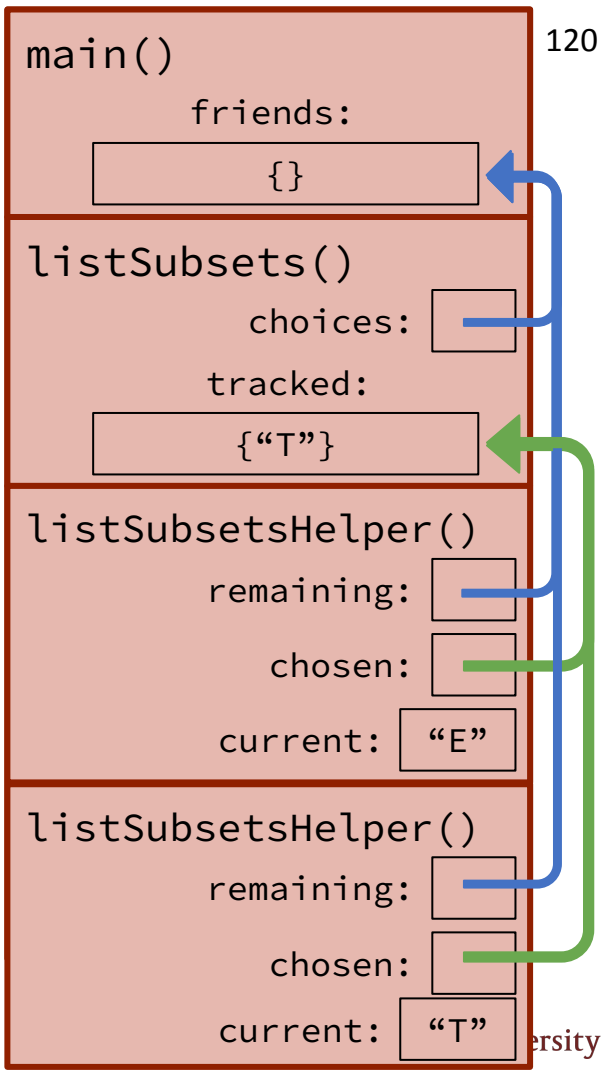
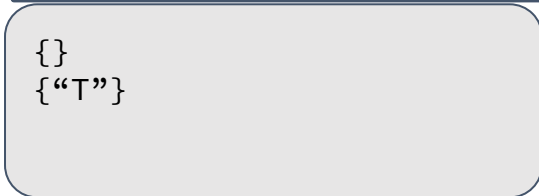
```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

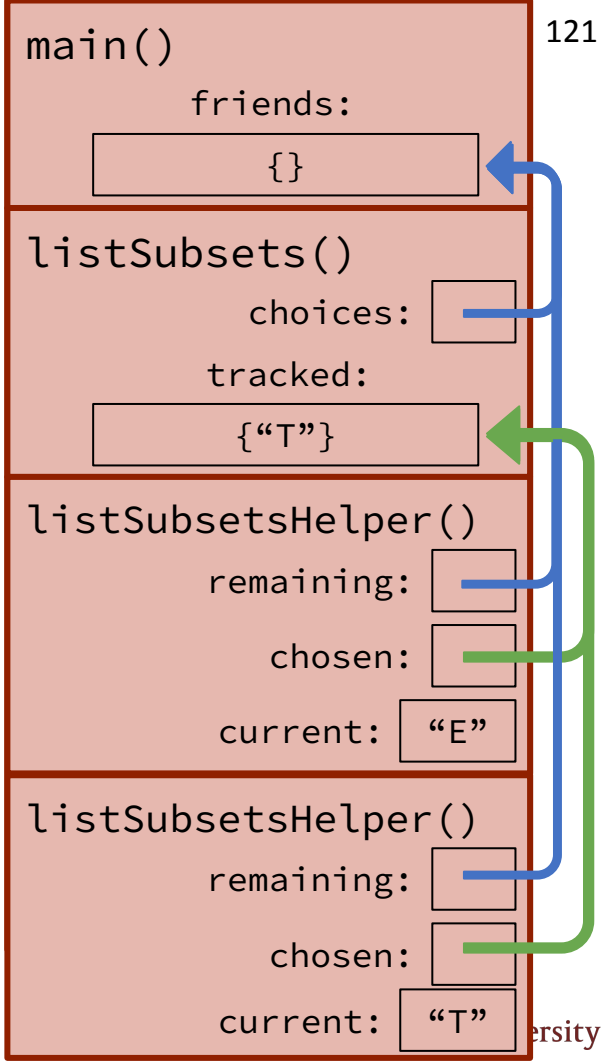
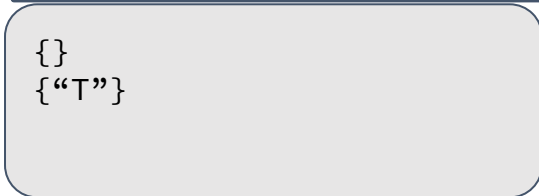
```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```




```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}

```

main()

122

friends:

{}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

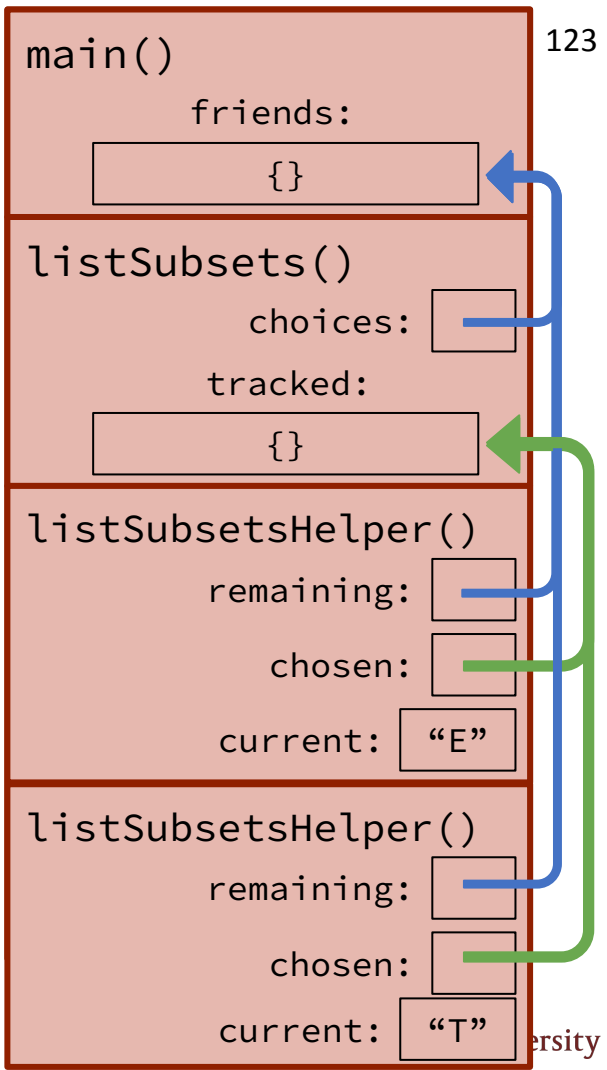
chosen:

current: "T"

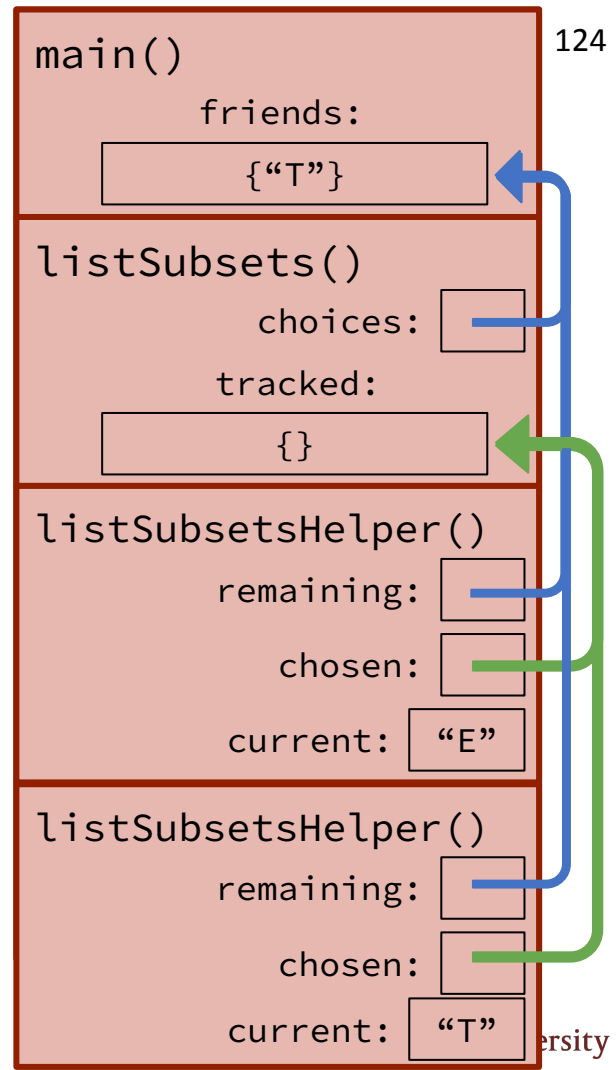
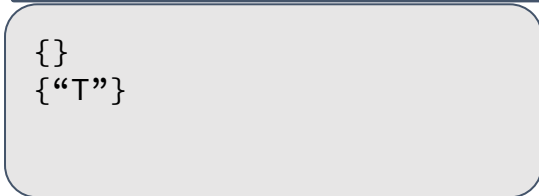
iversity

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{“T”}

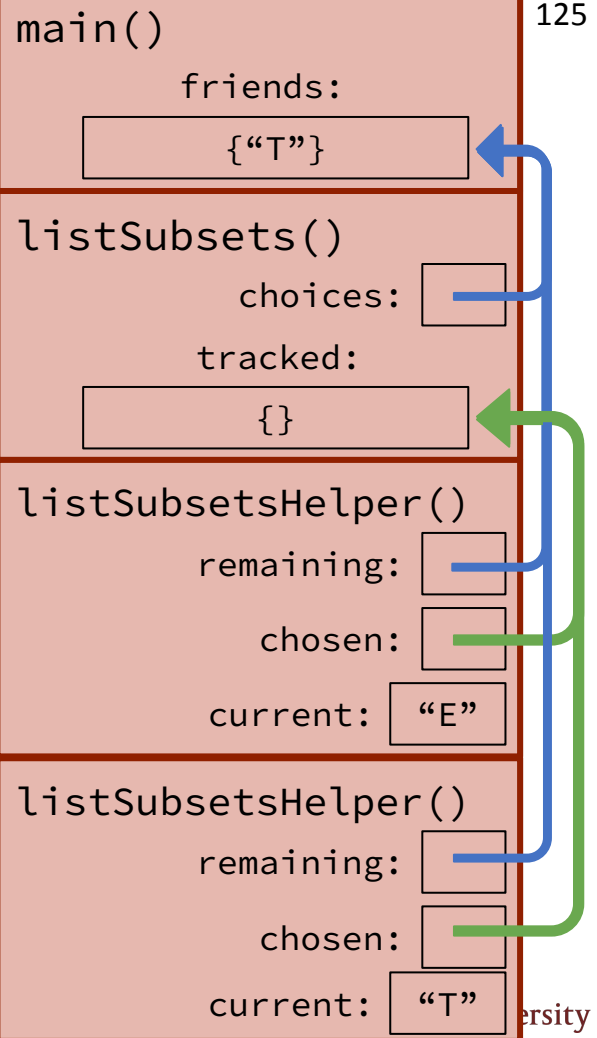


```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{“T”}



```

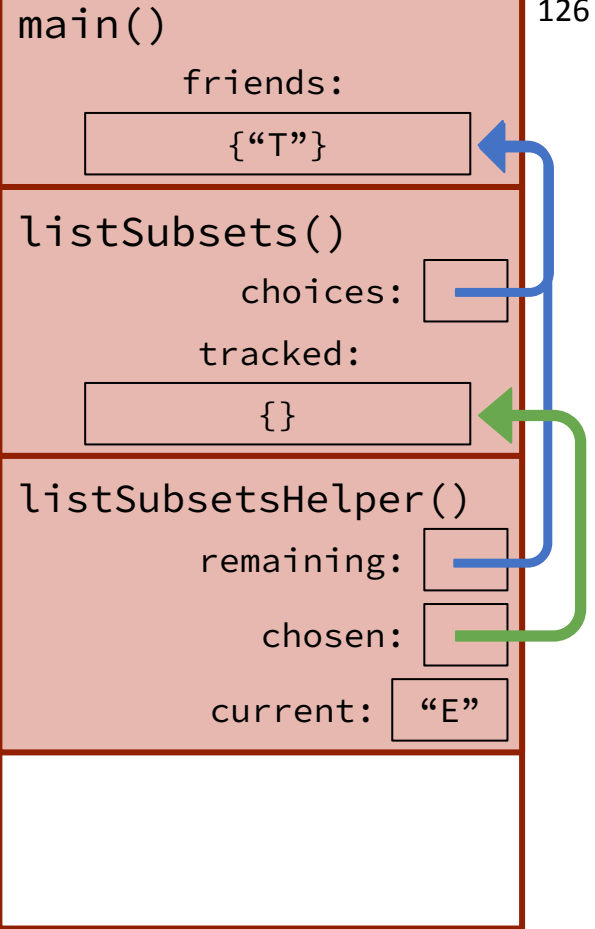
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}

```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}

```

main()

127

friends:

{"T"}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}

```

main()

128

friends:

{"T"}

listSubsets()

choices:

tracked:

{"E"}

listSubsetsHelper()

remaining:

chosen:

current: "E"


```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}

```

main()

129

friends:

{"T"}

listSubsets()

choices:

tracked:

{"E"}

listSubsetsHelper()

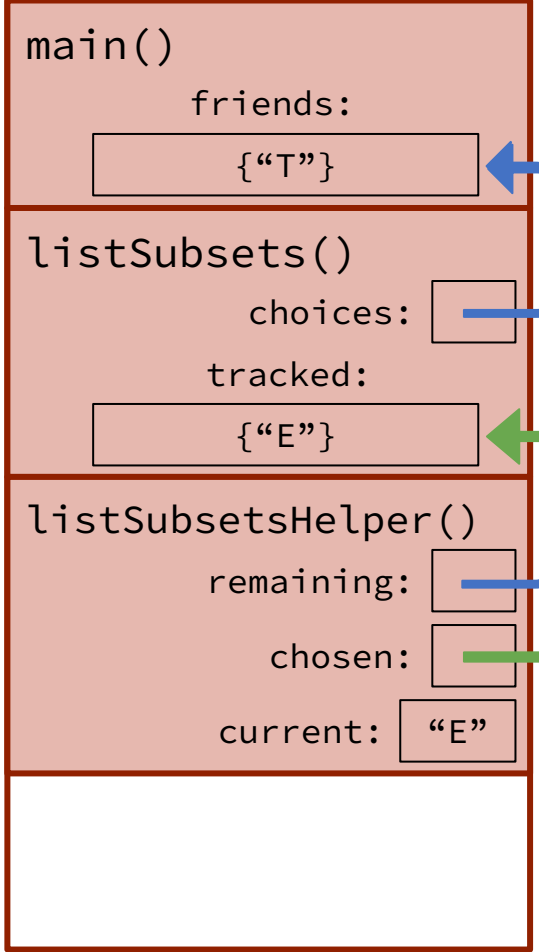
remaining:

chosen:

current: "E"

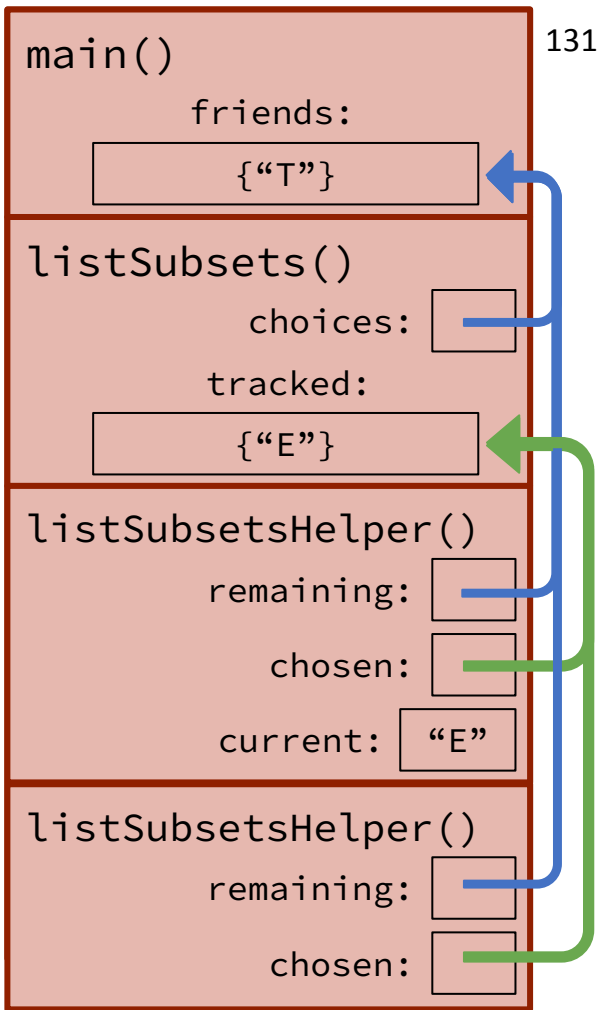
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }
{"T"}
```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }
{"T"}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}

```

main()

132

friends:

{"T"}

listSubsets()

choices:

tracked:

{"E"}

listSubsetsHelper()

remaining:

chosen:

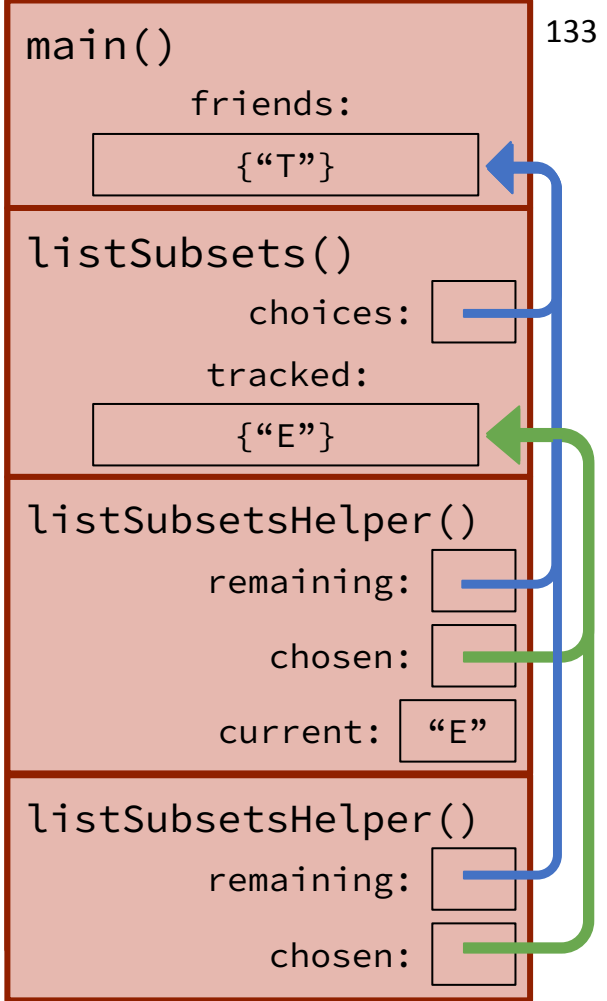
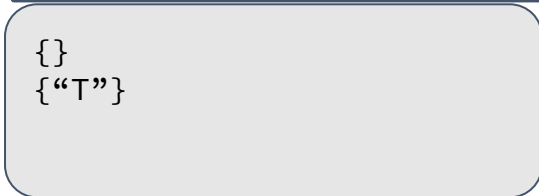
current: "E"

listSubsetsHelper()

remaining:

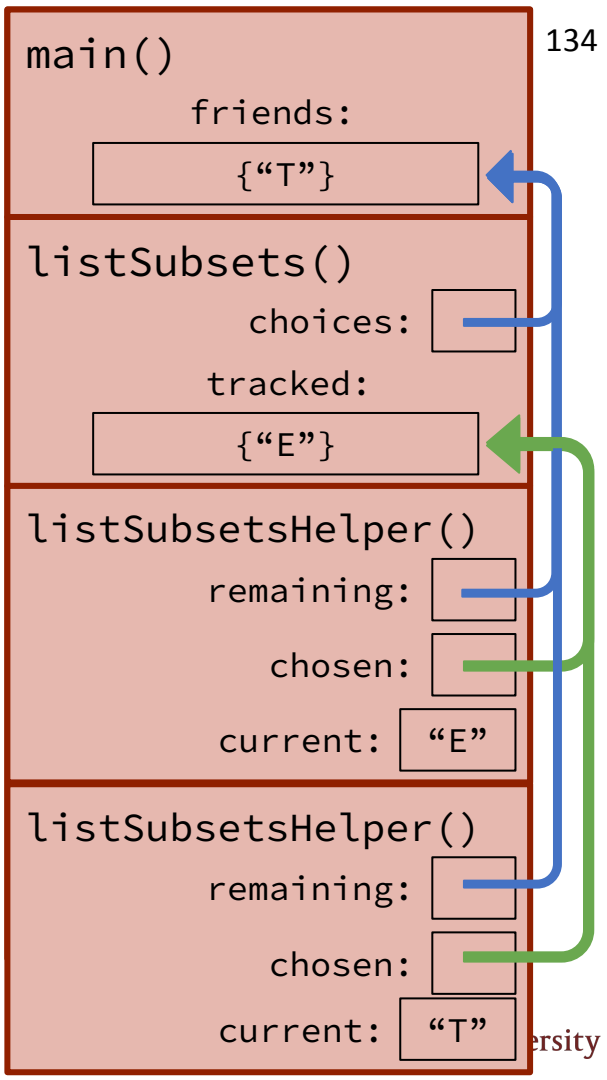
chosen:

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



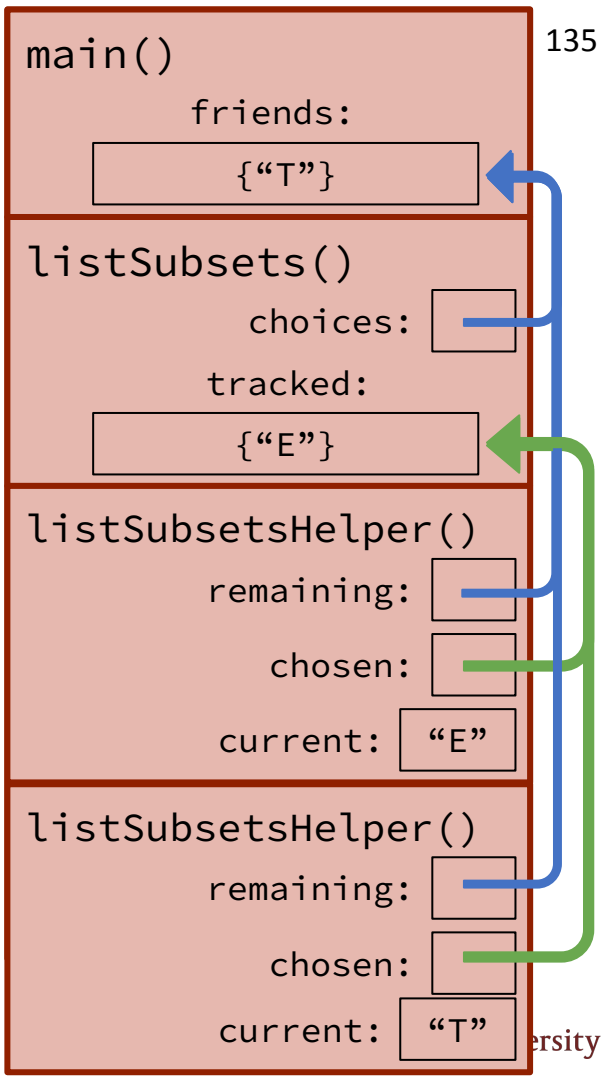
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }
{"T"}
```



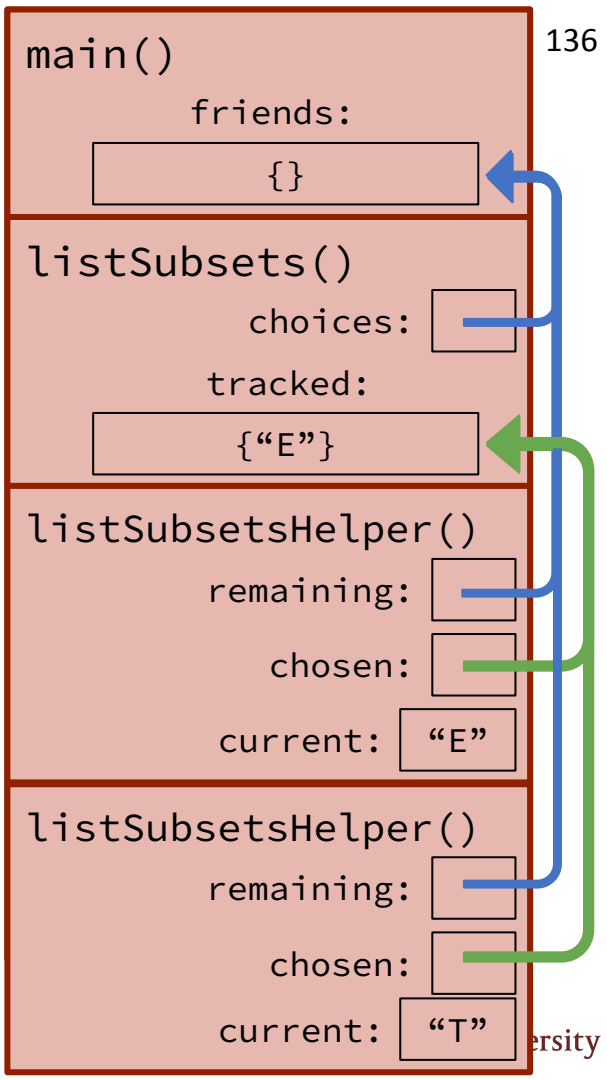
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{“T”}



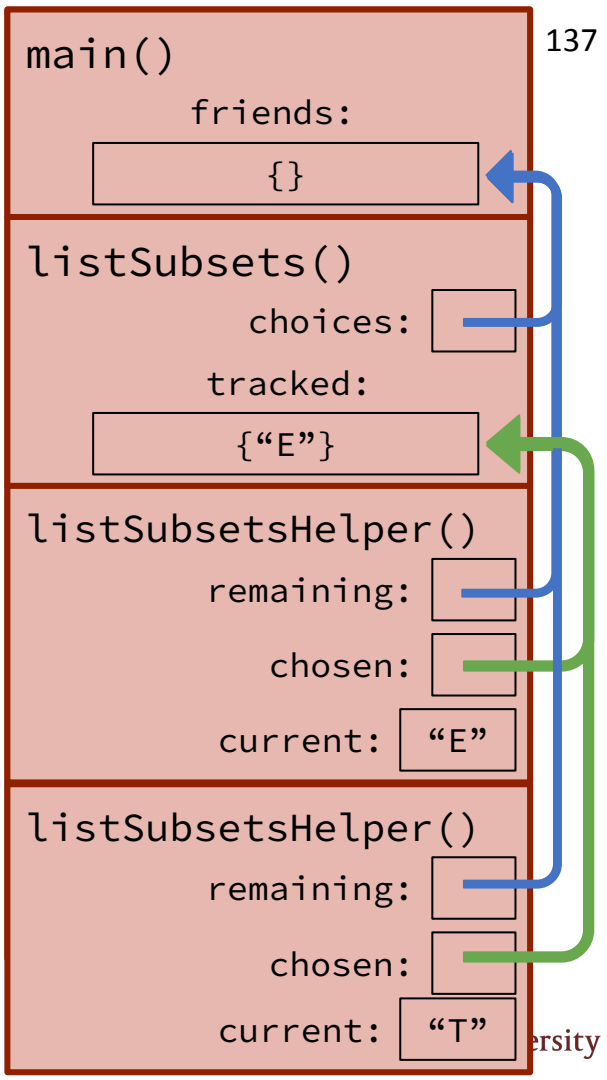
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{"T"}




```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{“T”}



```

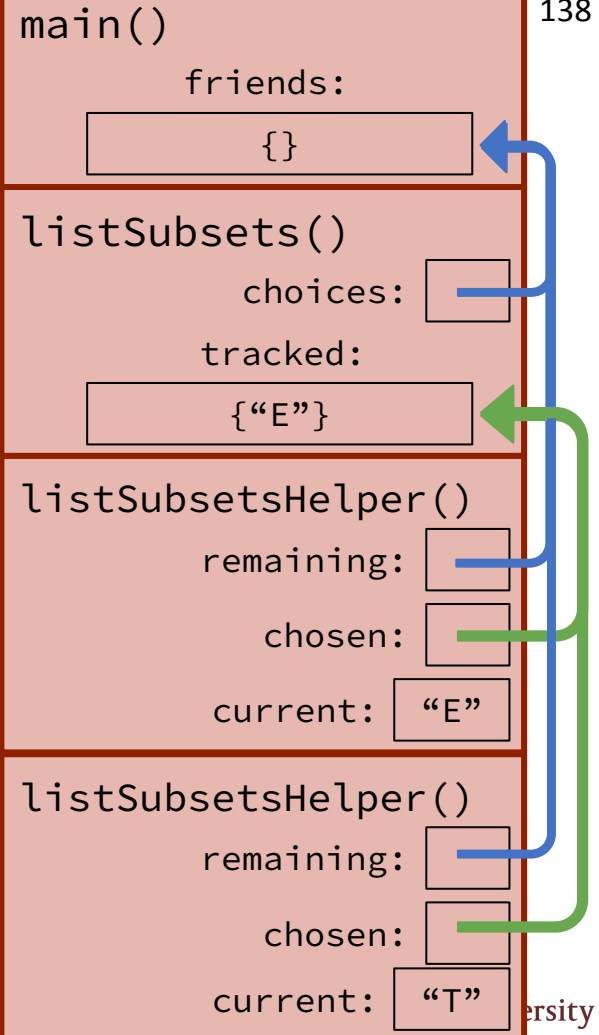
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

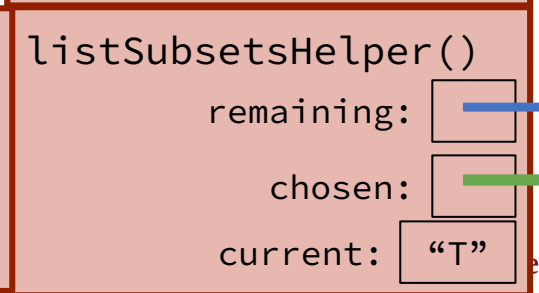
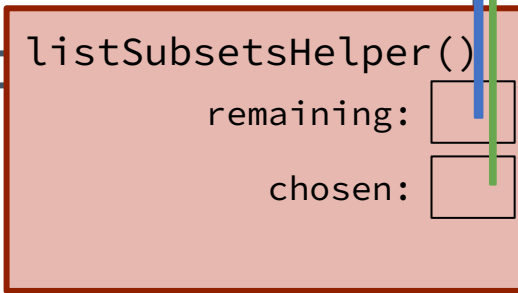
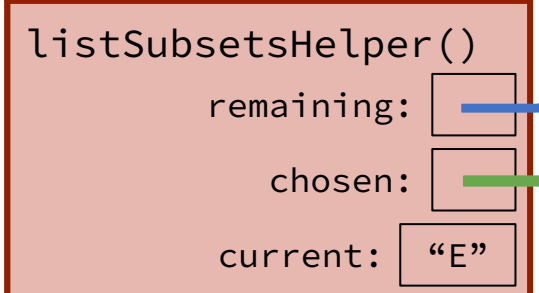
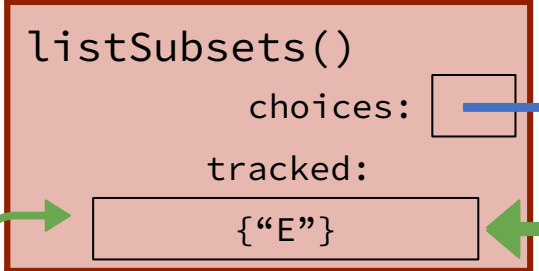
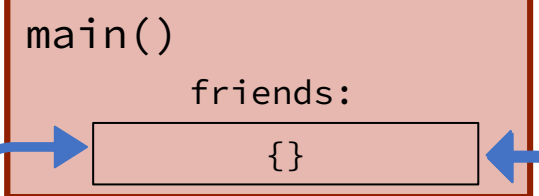
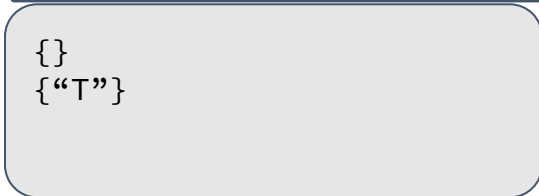
{}
{"T"}

```



```
void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
```

```
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }  
{"T"}
```

```
listSubsetsHelper()  
remaining: [ ]  
chosen: [ ]
```

main() 140

friends:

```
{ }
```

listSubsets()

choices: []

tracked:

```
{"E"}
```

listSubsetsHelper()

remaining: []

chosen: []

current: "E"

listSubsetsHelper()

remaining: []

chosen: []

current: "T"

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

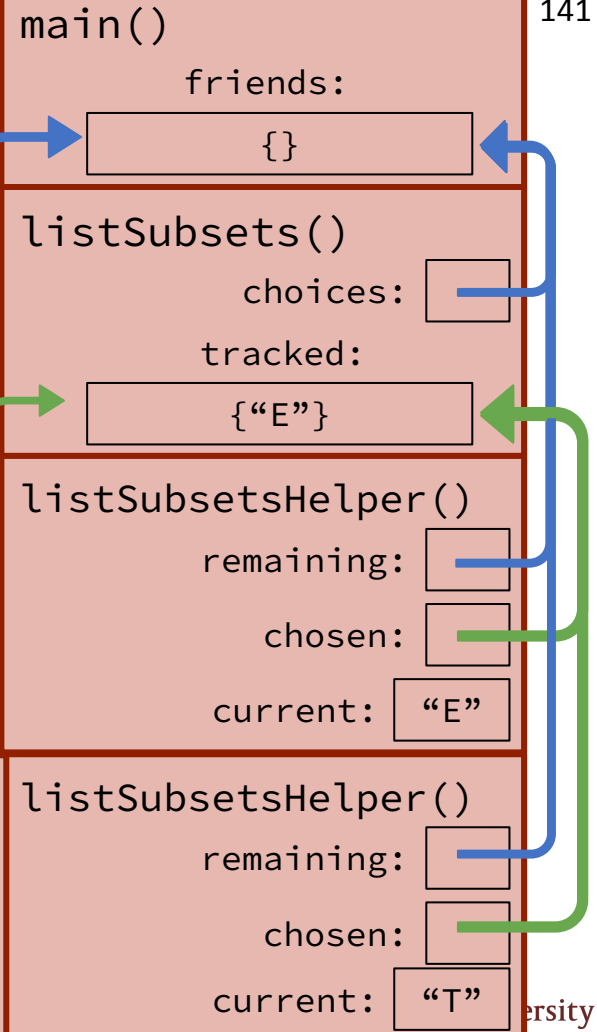
{}
{"T"}

```

```

listSubsetsHelper()
    remaining: [ ]
    chosen: [ ]

```



```

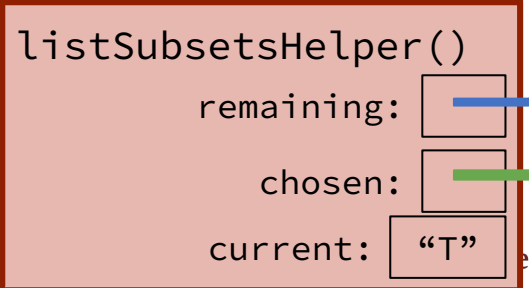
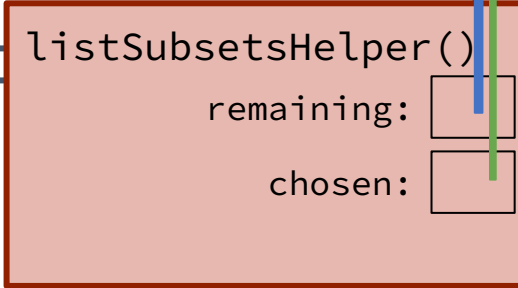
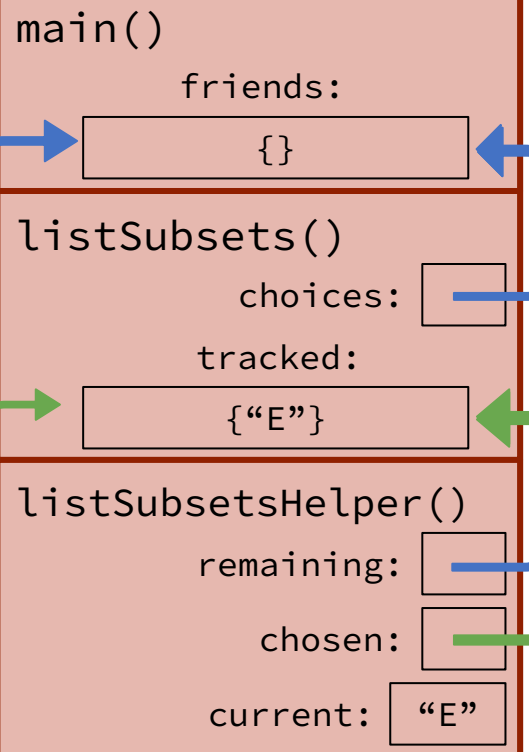
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}

```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}

```

main()

friends:

listSubsets()

choices:

tracked:

listSubsetsHelper()

remaining:

chosen:

current:

listSubsetsHelper()

remaining:

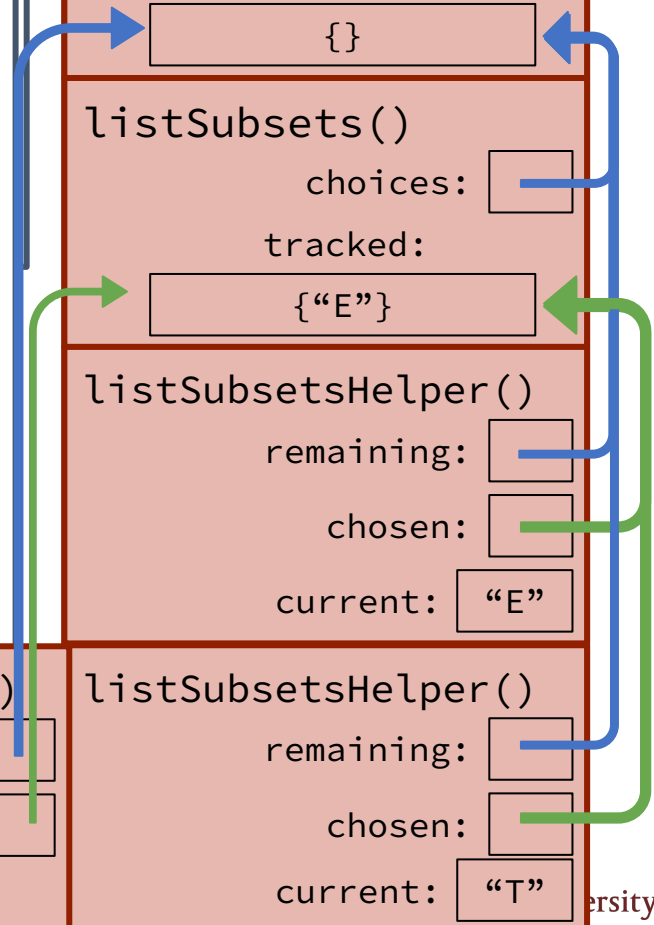
chosen:

listSubsetsHelper()

remaining:

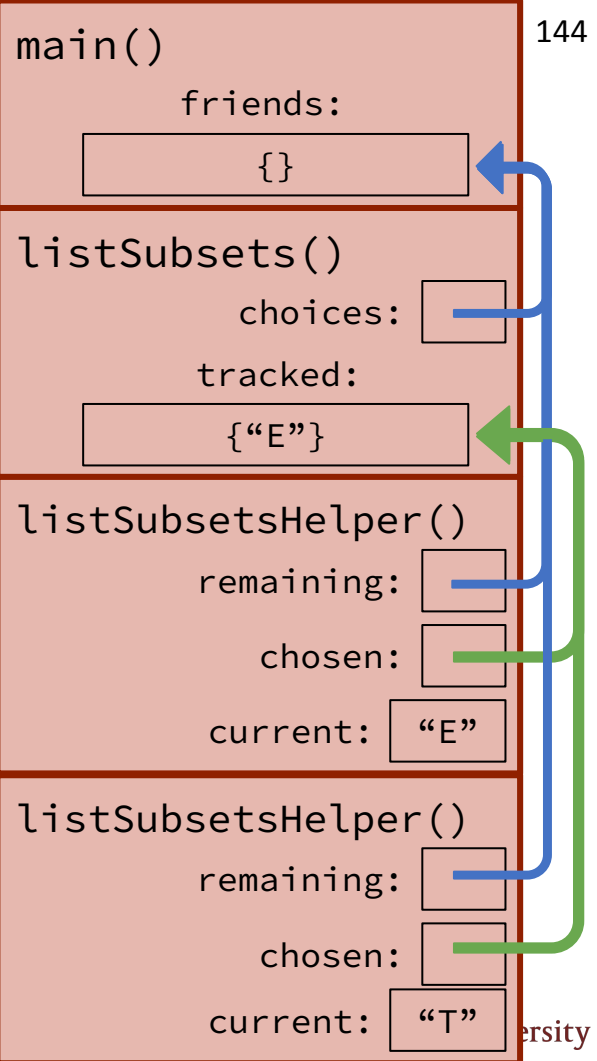
chosen:

current:



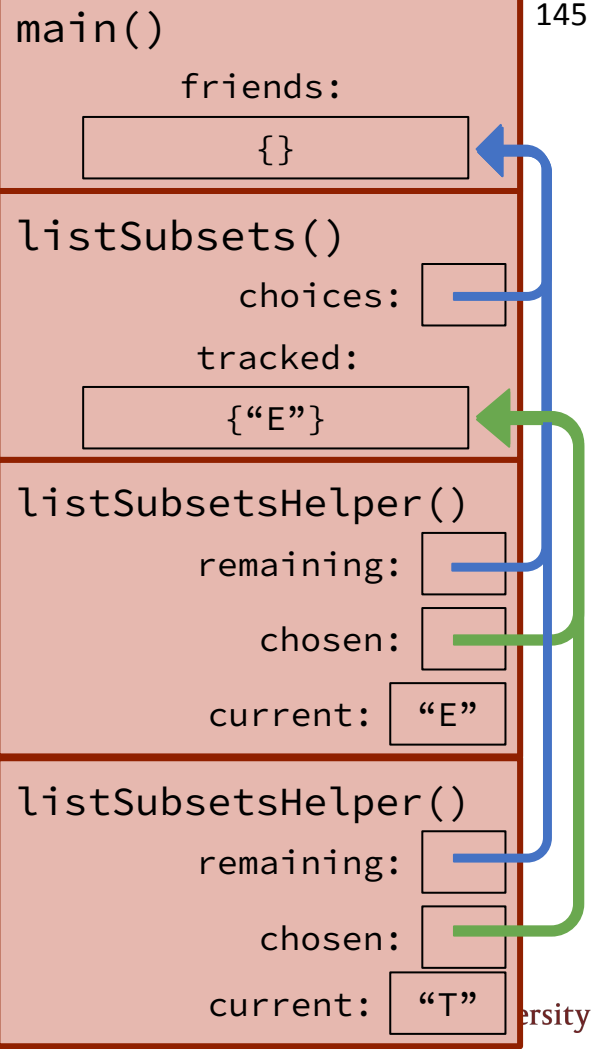
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{"T"}
{"E"}
}



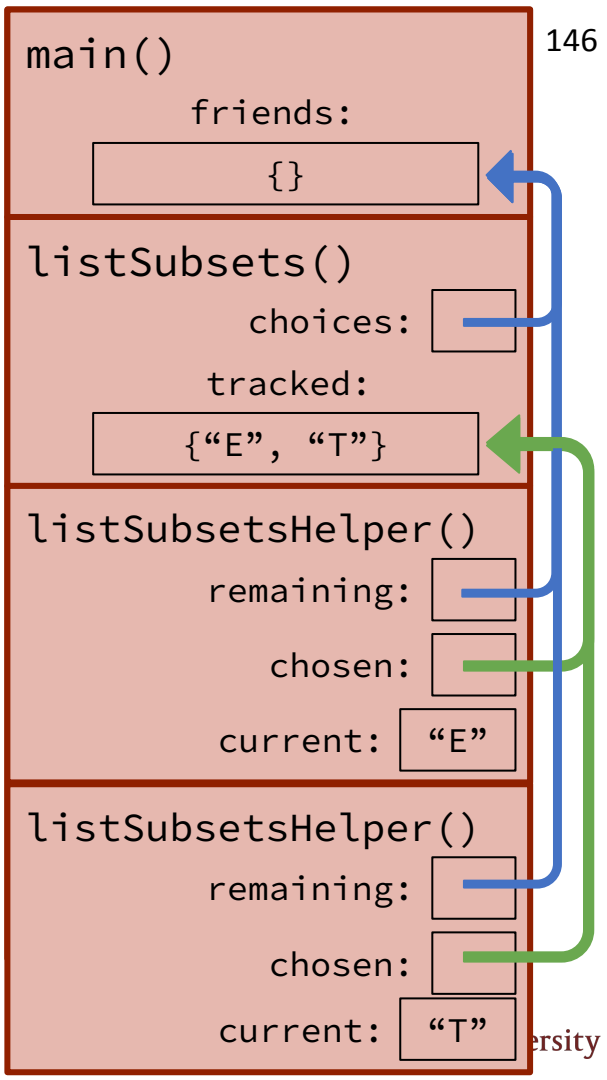

```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

{
{“T”}
{“E”}



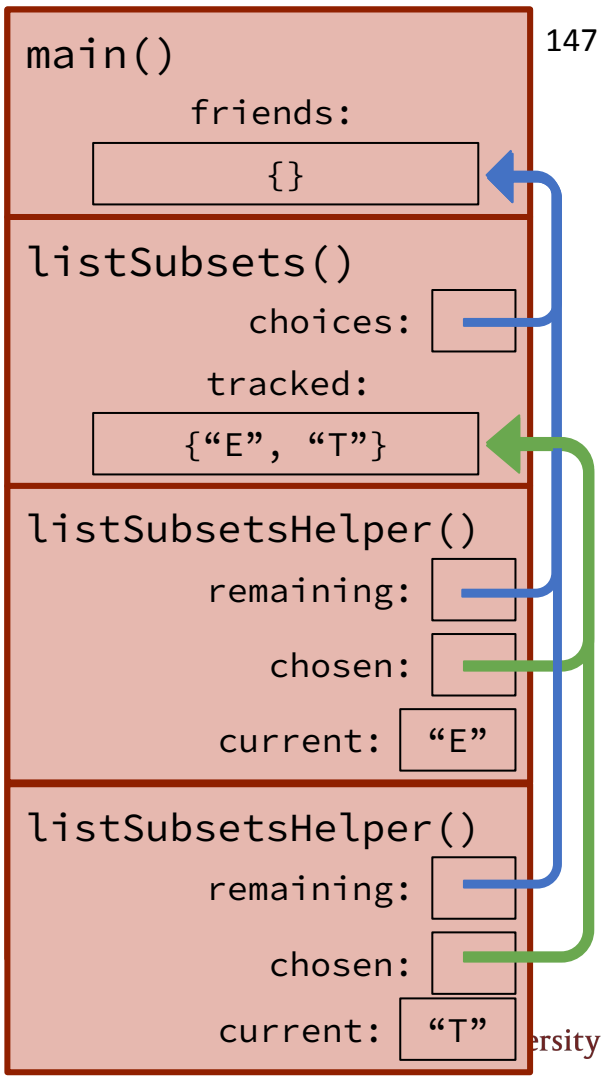
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }
{"T"}
{"E"}
{"E", "T"}
```



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }
{"T"}
{"E"}
{"E", "T"}
```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}

```

main()

148

friends:

{}

listSubsets()

choices:

tracked:

{"E", "T"}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

chosen:

current: "T"

ersity

```
void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
```

```
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

```
{ }
{ "T" }
{ "E" }
```

main()

friends:

listSubsets()

choices:

tracked:

listSubsetsHelper()

remaining:

chosen:

current:

listSubsetsHelper()

remaining:

chosen:

listSubsetsHelper()

remaining:

chosen:

current:

```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}

```

main()

friends:

listSubsets()

choices:

tracked:

listSubsetsHelper()

remaining:

chosen:

current:

listSubsetsHelper()

remaining:

chosen:

listSubsetsHelper()

remaining:

chosen:

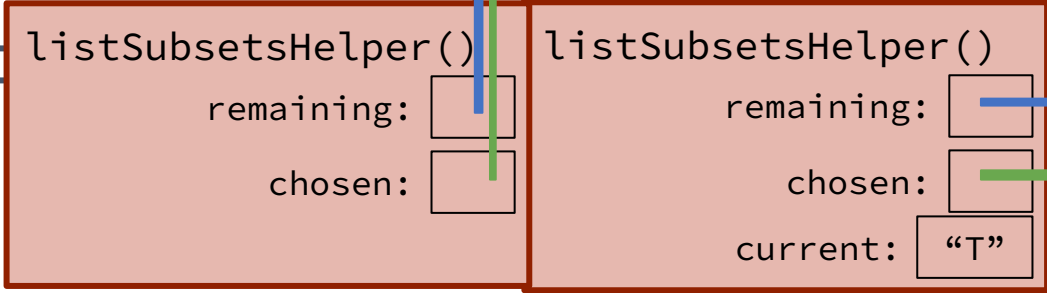
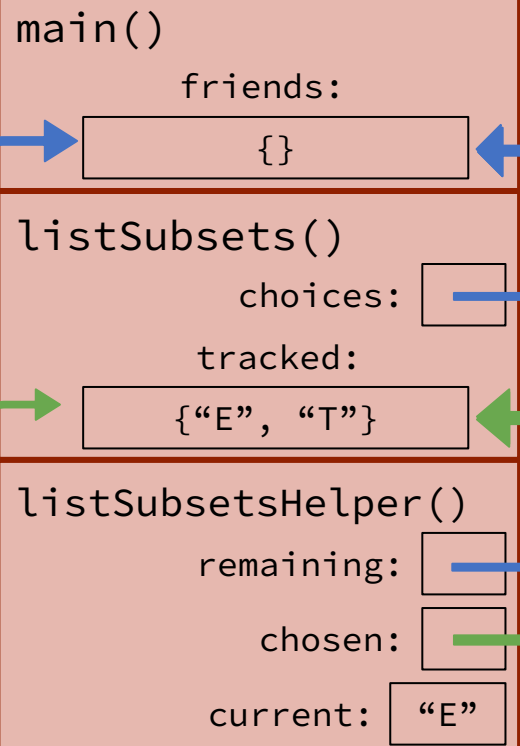
current:

```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

- {}
- {"T"}
- {"E"}



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

- {}
- {"T"}
- {"E"}
- {"E", "T"}

main()

friends:

listSubsets()

choices:

tracked:

listSubsetsHelper()

remaining:

chosen:

current:

listSubsetsHelper()

remaining:

chosen:

listSubsetsHelper()

remaining:

chosen:

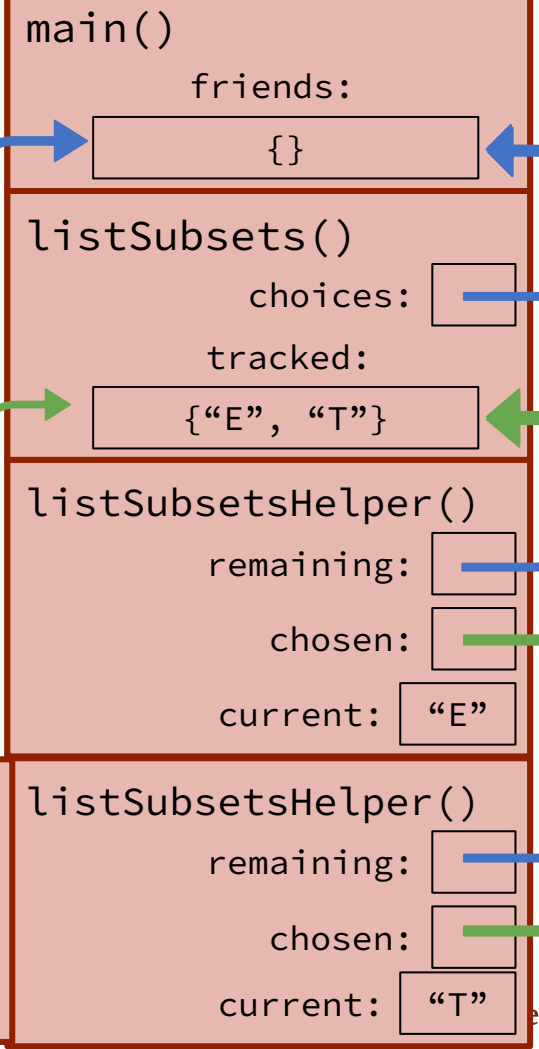
current:


```

void listSubsetsHelper(Set<string>& remaining,
                     Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

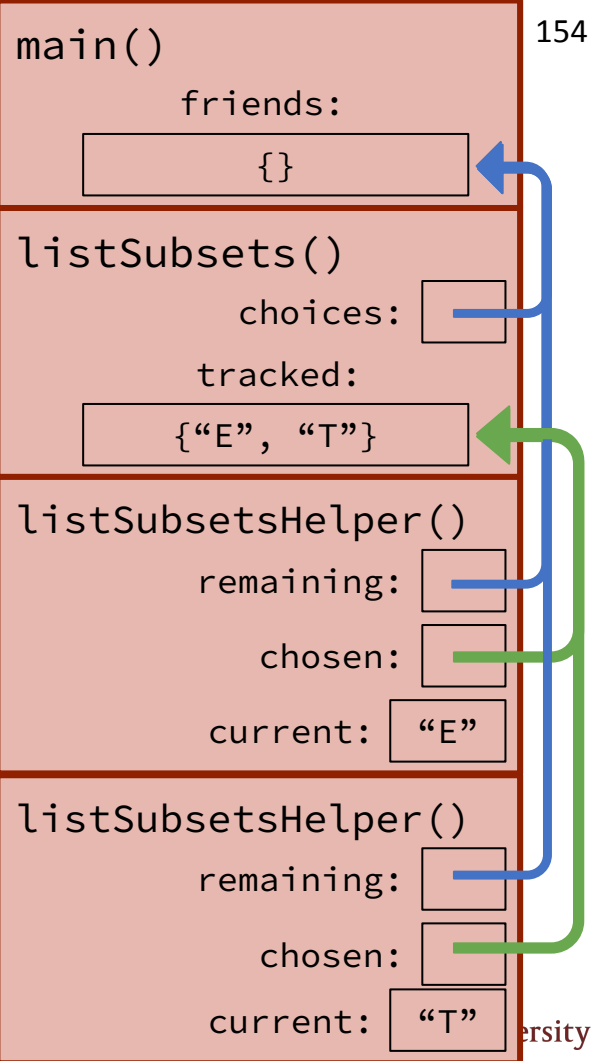
```

- {}
- {"T"}
- {"E"}
- {"E", "T"}



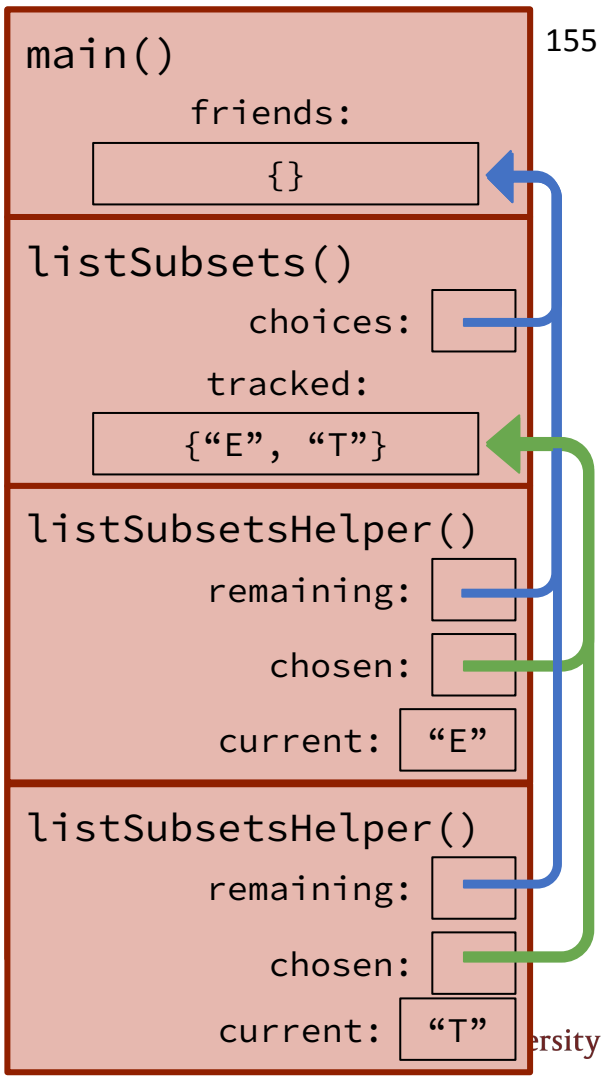
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

- {}
- {"T"}
- {"E"}
- {"E", "T"}



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

- {}
- {"T"}
- {"E"}
- {"E", "T"}



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```

main()

156

friends:

{}

listSubsets()

choices:

tracked:

{"E"}

listSubsetsHelper()

remaining:

chosen:

current: "E"

listSubsetsHelper()

remaining:

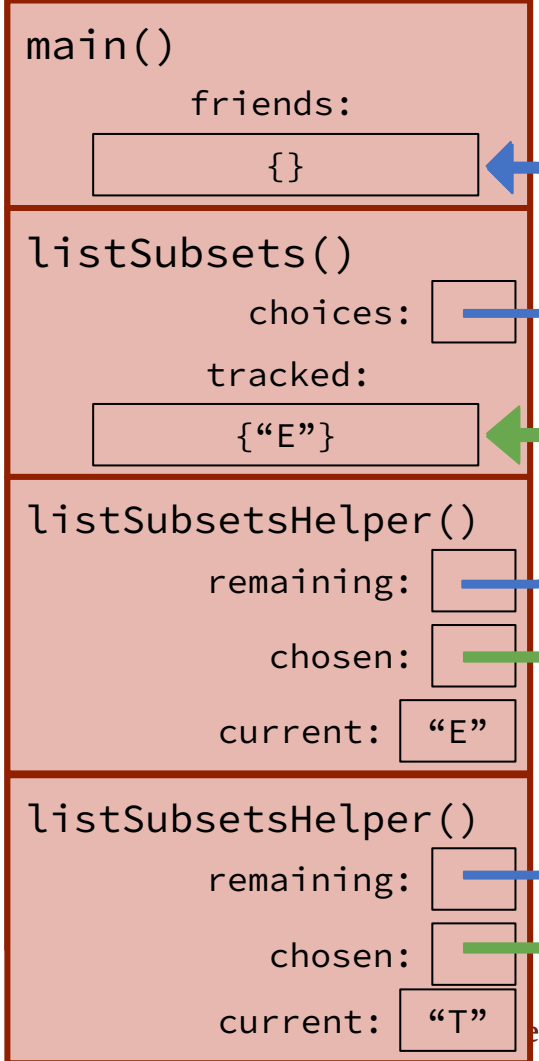
chosen:

current: "T"

iversity

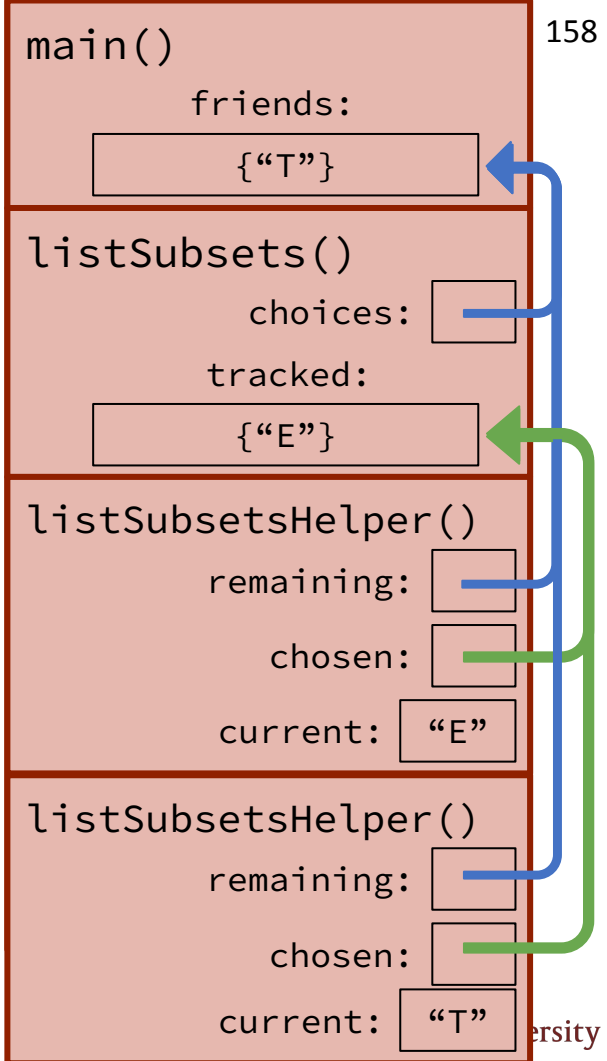
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

- {}
- {"T"}
- {"E"}
- {"E", "T"}



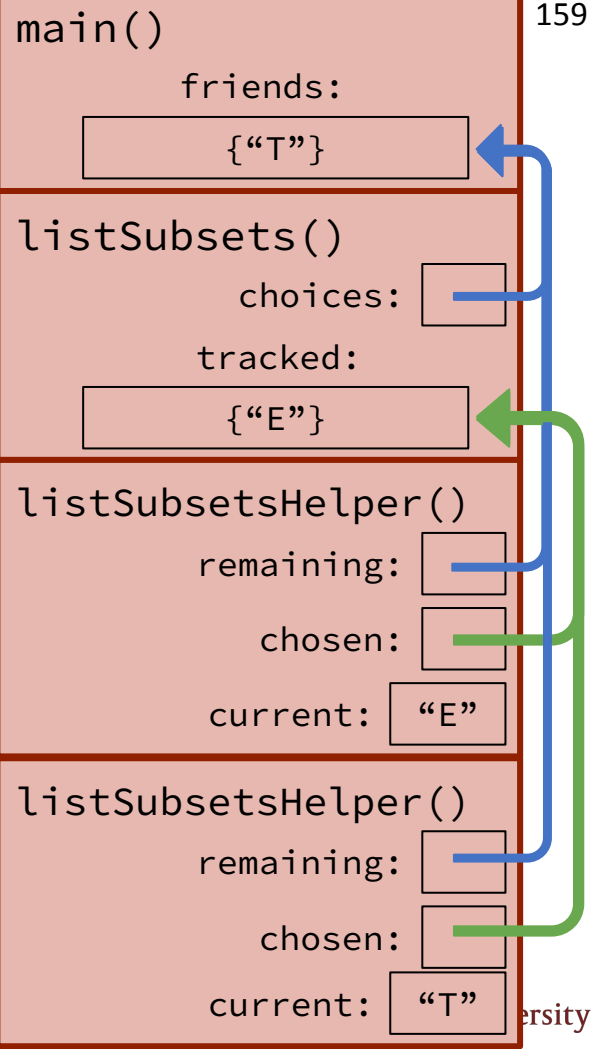
```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

- {}
- {"T"}
- {"E"}
- {"E", "T"}



```
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}
```

- {}
- {"T"}
- {"E"}
- {"E", "T"}



```

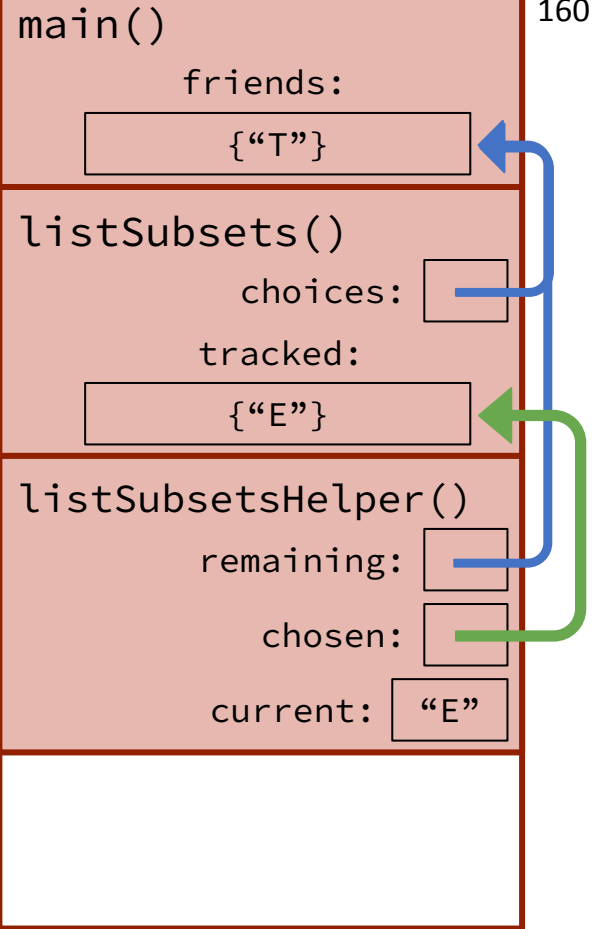
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```




```

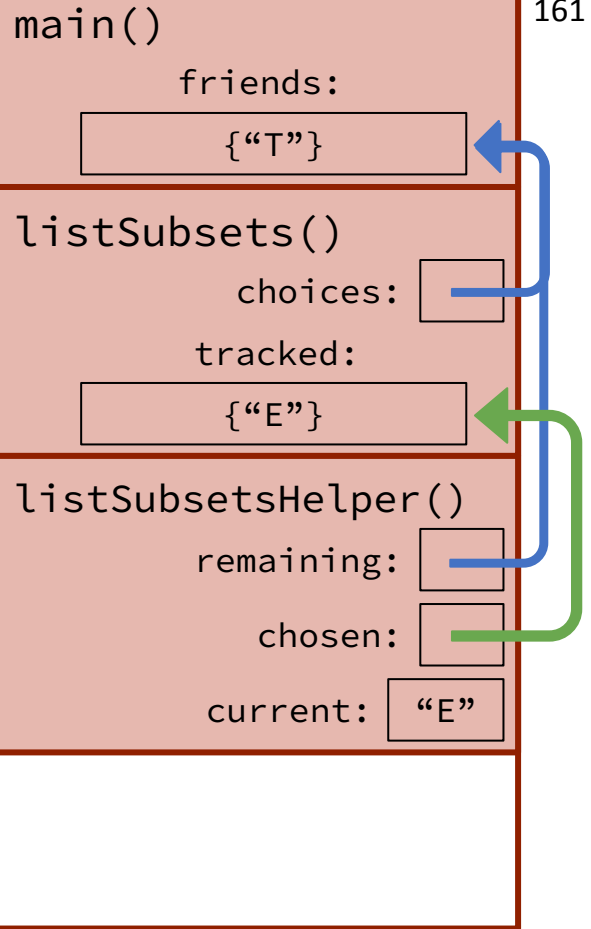
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```

main()

162

friends:

{"T"}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```

main()

163

friends:

{"T"}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: {"E"}

```

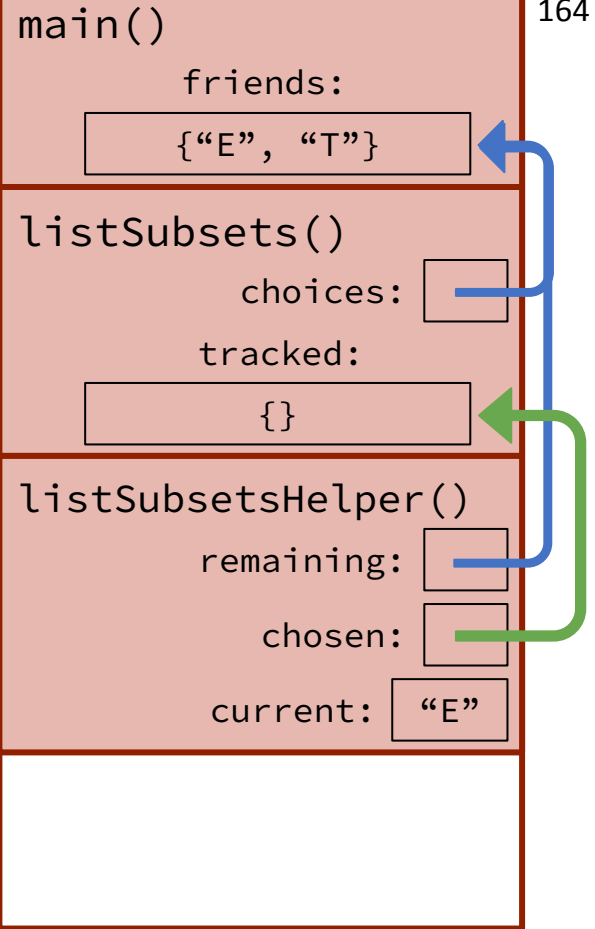
void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```



```

void listSubsetsHelper(Set<string>& remaining,
                      Set<string>& chosen) {
    if (remaining.isEmpty()) {
        cout << chosen << endl;
        return;
    }
    string current = remaining.first();
    remaining = remaining - current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen + current;
    listSubsetsHelper(remaining, chosen);
    chosen = chosen - current;
    remaining = remaining + current;
}

```

```

{}
{"T"}
{"E"}
{"E", "T"}

```

main()

165

friends:

{"E", "T"}

listSubsets()

choices:

tracked:

{}

listSubsetsHelper()

remaining:

chosen:

current: "E"

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

main()

166

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

{}
{“T”}
{“E”}
{“E”, “T”}

```
void listSubsets(Set<string>& choices) {  
    Set<string> tracked;  
    listSubsetsHelper(choices, tracked);  
}
```

```
{}  
{“T”}  
{“E”}  
{“E”, “T”}
```

main()

167

friends:

{“E”, “T”}

listSubsets()

choices:

tracked:

{}

```
int main () {  
    Set<string> friends = {'A', 'E', 'T'};  
    listSubsets(friends);  
    return 0;  
}
```

```
{}  
{“T”}  
{“E”}  
{“E”, “T”}
```

main()

friends:

```
{“E”, “T”}
```

168


```
int main () {  
    Set<string> friends = {'A', 'E', 'T'};  
    listSubsets(friends);  
    return 0;  
}
```

```
{}  
{“T”}  
{“E”}  
{“E”, “T”}
```

main()

friends:

```
{“E”, “T”}
```

169

{
{“T”}
{“E”}
{“E”, “T”}

Choose / explore / unchoose

- Implicit “unchoose” step
 - Pass by value; usually when memory constraints aren’t an issue
 - Works because you’re making edits to a copy
 - E.g. Building up a string over time
- Explicit “unchoose” step
 - Uses pass by reference; usually with large data structures
 - “Undoing” prior modifications to structure
 - E.g. Generating subsets (one set passed around by reference to track subsets)

Mazes (Revisited)

Solving Recursive Backtracking

- **Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, or pick one best solution)**
- What's the provided function prototype and requirements? Do we need a helper function?
 - What are we returning as our solution?
 - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?
- What are our base and recursive cases?
 - What does the decision tree look like? (decisions, options, what to keep track of)
 - In addition to what we're building up, are there any additional constraints on our solutions?
 - Does it make sense to use an implicit or explicit unchoose step for the recursion?

Solving Recursive Backtracking

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, or pick one best solution)
- **What's the provided function prototype and requirements? Do we need a helper function?**
 - **What are we returning as our solution?**
 - **Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?**
- What are our base and recursive cases?
 - What does the decision tree look like? (decisions, options, what to keep track of)
 - In addition to what we're building up, are there any additional constraints on our solutions?
 - Does it make sense to use an implicit or explicit unchoose step for the recursion?

Do we need a helper function?

- Recall our function prototype:

```
Vector<GridLocation> solveMazeDFS(Grid<bool>& maze);
```

Do we need a helper function?

- Recall our function prototype:

```
Vector<GridLocation> solveMazeDFS(Grid<bool>& maze);
```

- We need a helper function to keep track of our path through the maze!
 - Our helper function will have as parameters: the maze itself and the path we're building up.
 - We also want the helper to be able to tell us whether or not the maze is solvable – let's have it return a boolean.

Do we need a helper function?

- Recall our function prototype:

```
Vector<GridLocation> solveMazeDFS(Grid<bool>& maze);
```

- We need a helper function to keep track of our path through the maze!
 - Our helper function will have as parameters: the maze itself and the path we're building up.
 - We also want the helper to be able to tell us whether or not the maze is solvable – let's have it return a boolean.

```
bool solveMazeHelper(Grid<bool>& maze,  
                    Stack<GridLocation>& path)
```

Solving Recursive Backtracking

- Which of our three use cases does our problem fall into? (generate/count all solutions, find one solution/prove its existence, or pick one best solution)
- What's the provided function prototype and requirements? Do we need a helper function?
 - What are we returning as our solution?
 - Do we care about returning or keeping track of the path we took to get to our solution? If yes, what parameters are we already given and what others might be useful?
- **What are our base and recursive cases?**
 - **What does the decision tree look like? (decisions, options, what to keep track of)**
 - **In addition to what we're building up, are there any additional constraints on our solutions?**
 - **Does it make sense to use an implicit or explicit unchoose step for the recursion?**

Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, West
- Repeat until we're at the end of the maze

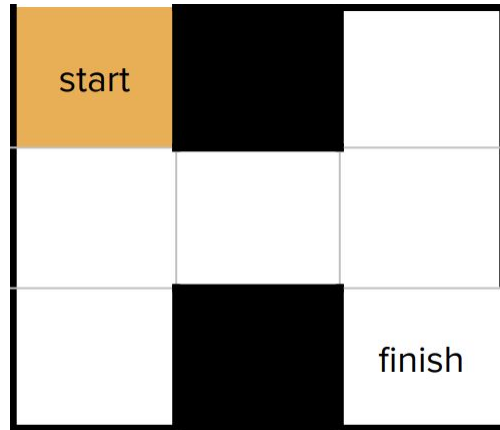
Recursive Algorithm

- Start at the entrance



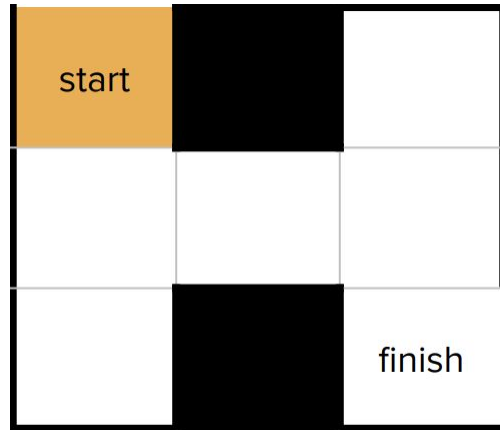
Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, or West



Recursive Algorithm

- Start at the entrance
- Take one step ~~North~~, South, East, or West



Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Recursive Algorithm

- Start at the entrance
- Take one step ~~North~~, South, East, or West



Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Recursive Algorithm

- Start at the entrance
- Take one step ~~North, South, East, or West~~

Dead end!



Recursive Algorithm

- Start at the entrance
- Take one step ~~North, South, East, or West~~

We have to go
back one step



Recursive Algorithm

- Start at the entrance
- Take one step ~~North~~, South, East, or West



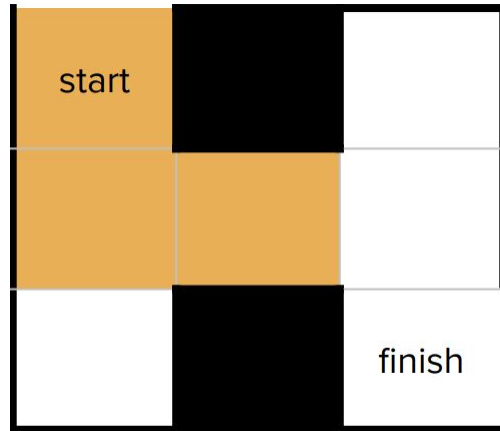
Recursive Algorithm

- Start at the entrance
- Take one step ~~North, South~~, East, or West



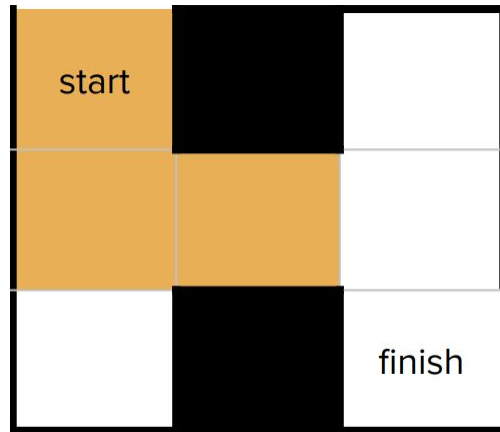
Recursive Algorithm

- Start at the entrance
- Take one step ~~North, South~~, East, or West
- Repeat until we're at the end of the maze



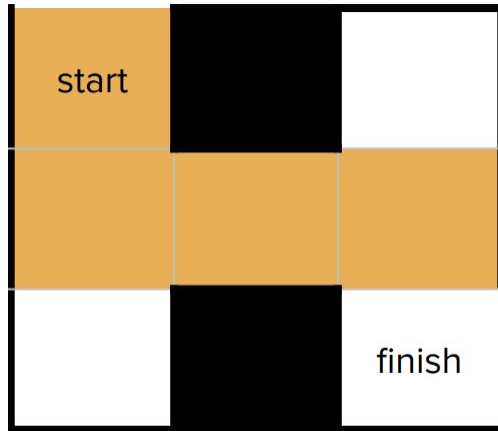
Recursive Algorithm

- Start at the entrance
- Take one step ~~North, South~~, East, or West



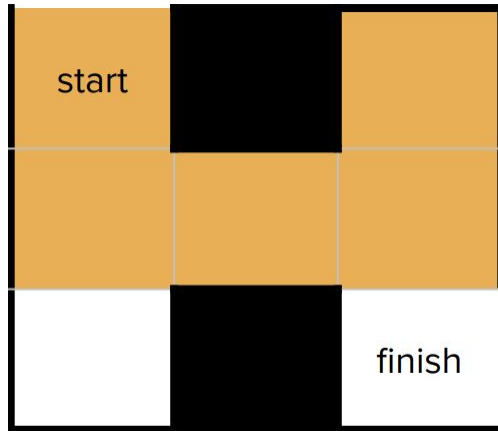
Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Recursive Algorithm

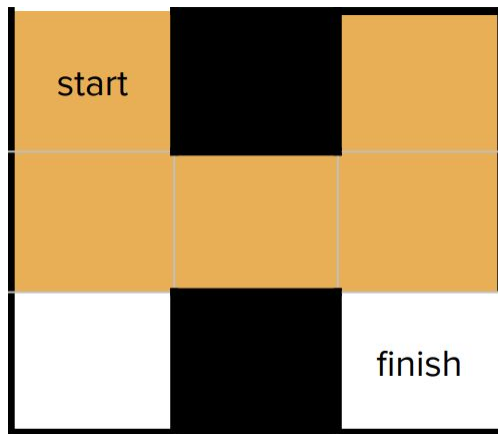
- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Recursive Algorithm

- Start at the entrance
- Take one step ~~North, South, East, or West~~

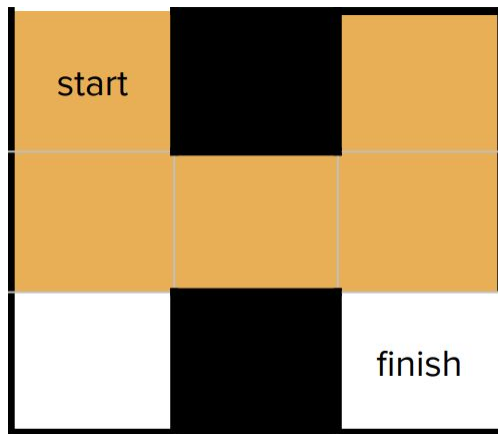
Dead end!



Recursive Algorithm

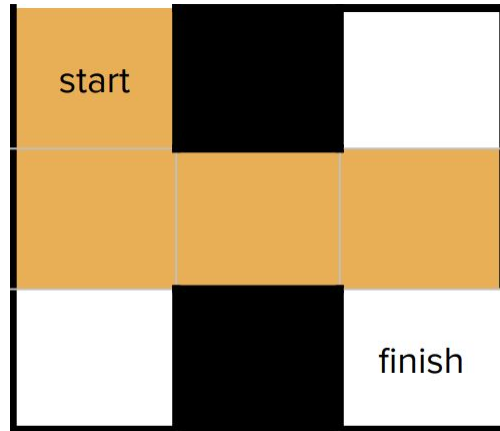
- Start at the entrance
- Take one step ~~North, South, East, or West~~

We have to go
back one step



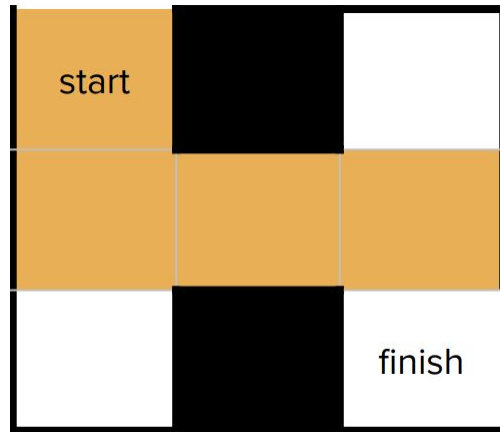
Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, or West



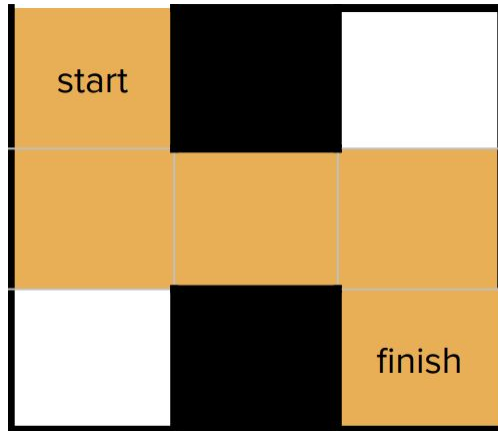
Recursive Algorithm

- Start at the entrance
- Take one step ~~North~~, South, East, or West



Recursive Algorithm

- Start at the entrance
- Take one step North, South, East, or West
- Repeat until we're at the end of the maze



Recursive Algorithm

- Base case:
- Recursive case:



Recursive Algorithm

- Base case: If we're at the end of the maze, stop
- Recursive case: Explore North, South, East, then West



Making a Decision Tree

- Decision at each step (each level of the tree)
 - Which valid move will we take?
- Options at each decision (branches from each node)
 - All valid moves (in bounds, not a wall, not previously visited) that are either North, South, East, or West of the current location
- Information you need to store along the way
 - The path we've taken so far (a Vector we're building up)
 - Where we've already visited
 - **Our current location**

Do we need a helper function?

- Recall our function prototype:

```
Vector<GridLocation> solveMazeDFS(Grid<bool>& maze);
```

- We need a helper function to keep track of our path through the maze!
 - Our helper function will have as parameters: the maze itself and the path we're building up.
 - We also want the helper to be able to tell us whether or not the maze is solvable – let's have it return a boolean.

```
bool solveMazeHelper(Grid<bool>& maze,  
                    Stack<GridLocation>& path,  
                    GridLocation cur)
```

Pseudocode

- Our helper function will have as parameters: the maze itself, the path we're building up, and the current location.
 - Idea: Use the boolean Grid (the maze itself) to store information about whether or not a location has been visited by flipping the cell to false once it's in the path (to avoid loops) → This works with our existing `generateValidMoves()` function

Pseudocode

- Our helper function will have as parameters: the maze itself, the path we're building up, and the current location.
 - Idea: Use the boolean Grid (the maze itself) to store information about whether or not a location has been visited by flipping the cell to false once it's in the path (to avoid loops) → This works with our existing generateValidMoves() function
- Recursive case: Iterate over valid moves from generateValidMoves() and try adding them to our path
 - If any recursive call returns true, we have a solution
 - If all fail, return false

Pseudocode

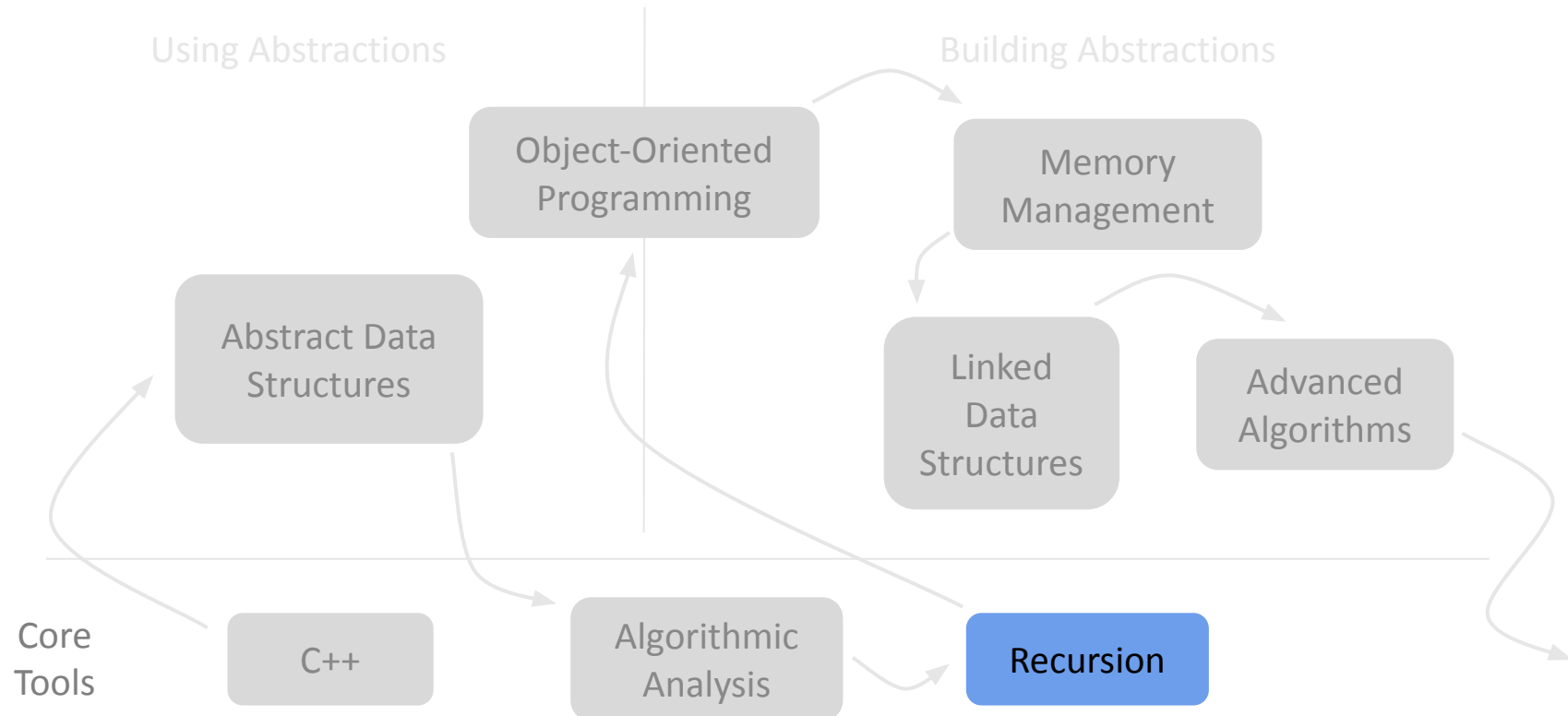
- Our helper function will have as parameters: the maze itself, the path we're building up, and the current location.
 - Idea: Use the boolean Grid (the maze itself) to store information about whether or not a location has been visited by flipping the cell to false once it's in the path (to avoid loops) → This works with our existing generateValidMoves() function
- Recursive case: Iterate over valid moves from generateValidMoves() and try adding them to our path
 - If any recursive call returns true, we have a solution
 - If all fail, return false
- Base case: We can stop exploring when we've reached the exit → return true if the current location is the exit

Let's Code It Up!

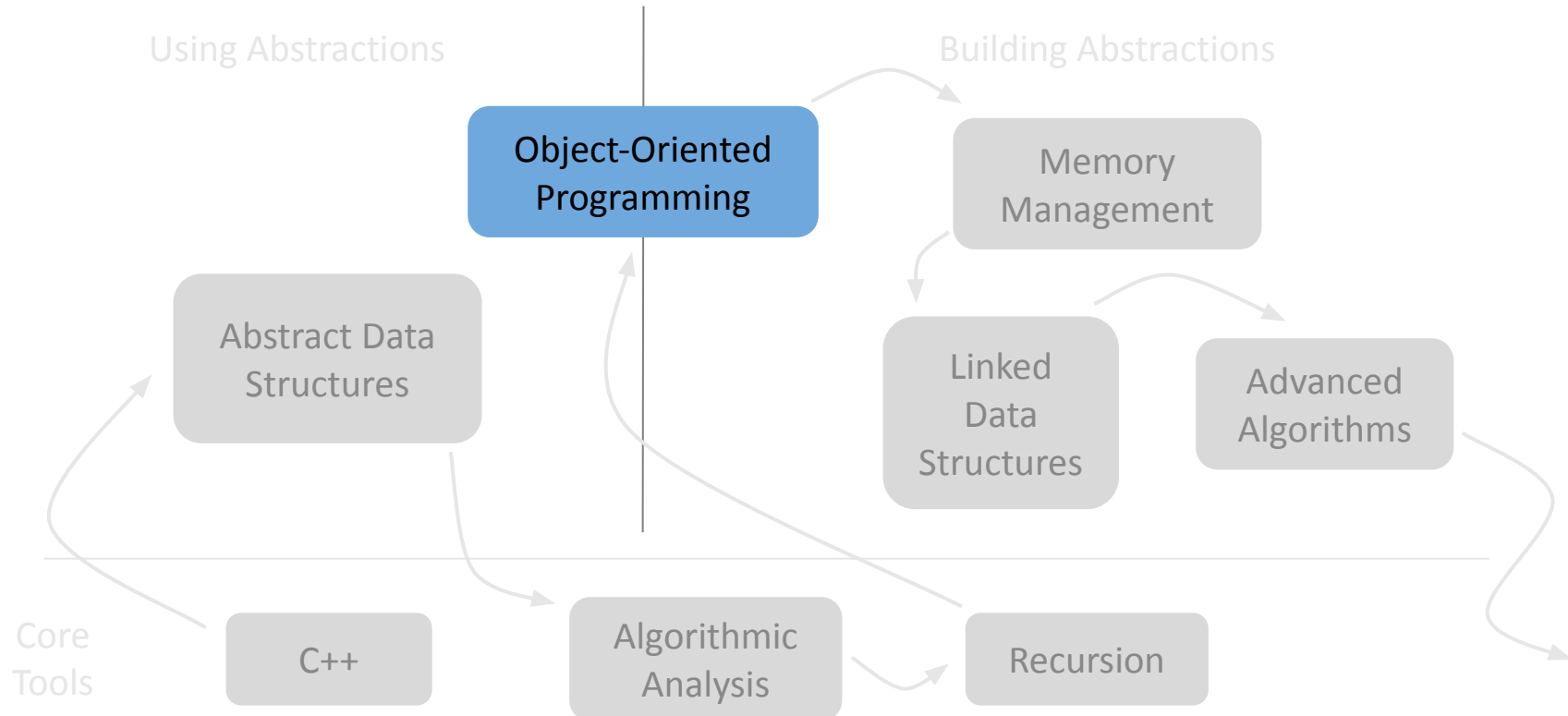
Takeaways

- Recursive maze-solving uses explicit “unchoose” because we have to set cells back to true after trying them.
- Our helper function may have a different return type from our initial function prototype, and our wrapper function (not the helper) may be more complex than just a call to our helper function.
- It may be helpful to revisit and adjust our initial answers to our planning questions as we determine more about the algorithm we want to use (e.g. adding a parameter to our helper function).

Roadmap



Roadmap



Const References

Let's Compare

Pass by value

- Callee gets a copy of a variable from the caller function
- Changes to that variable that occur in callee do not persist in caller



Pass by reference

- Callee gets a **reference** to a variable from the caller function
- Now, the callee can directly modify the original variable

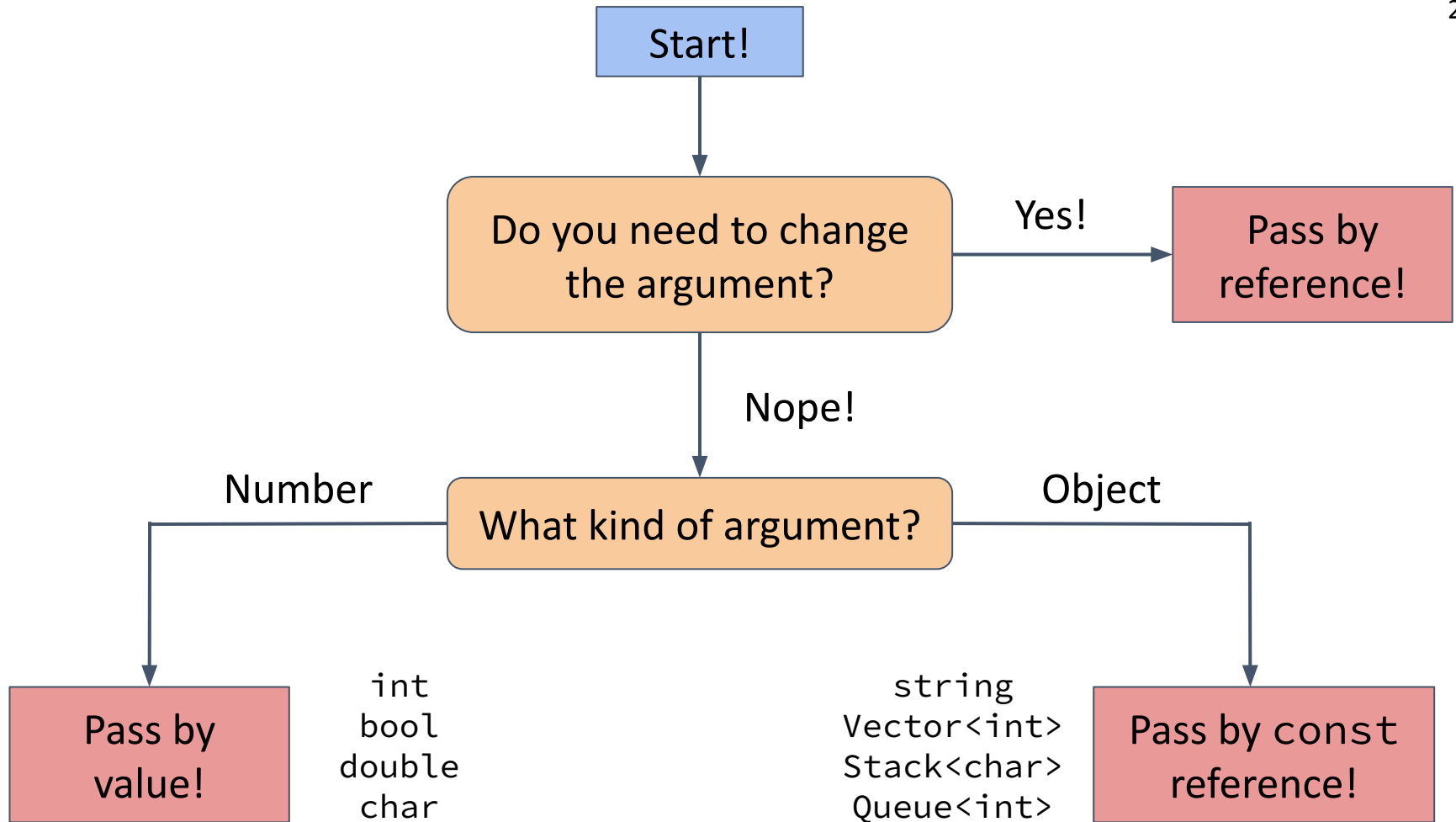


Pass by Reference

- Pros:
 - Don't have to make an expensive copy
 - E.g. Don't have to handwrite a copy of our essay
- Cons:
 - Callee can directly modify my copy
 - E.g. Friend can edit your copy of the essay

Pass by Const Reference

- If you want to look at, but not modify, a function parameter, pass it by **const reference**
- The “by reference” part avoids a copy.
- The “const” (constant) part means that the function can’t change that argument.



Structs

Struct


- Way to bundle different types of information
 - Package data into one place
- Like creating a custom data structure or variable

GridLocation struct

```
struct GridLocation {  
    int row;  
    int col;  
};
```

GridLocation struct

```
struct GridLocation {  
    int row;  
    int col;  
};
```



struct definition

GridLocation struct

```
struct GridLocation {  
    int row;  
    int col;  
};
```

} struct members

GridLayout struct

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

GridLocation struct

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

```
GridLocation origin = {0, 0};
```

GridLocation struct

- To declare a struct, you can either assign each of its members separately or assign it when it's created:

```
GridLocation origin = {0, 0};
```

```
GridLocation origin;
```

```
origin.row = 0;
```

```
origin.col = 0;
```

Examples of structs

- Store information about dates, but C++ does not have a Date type

```
struct Date {  
    int year;  
    int month;  
    int day;  
};
```

Examples of structs

- Store information about what is in my Lunchable

```
struct Lunchable {  
    string dessert;  
    int numCrackers;  
    bool hasCheese;  
};
```


Examples of structs

- Store information about albums I like

```
struct Album {  
    string title;  
    int year;  
  
    string artist_name;  
    int artist_age;  
    string artist_favorite_food;  
    int artist_height;  
};
```

Examples of structs

- Store information about albums I like

```
struct Album {  
    string title;  
    int year;  
  
    string artist_name;  
    int artist_age;  
    string artist_favorite_food;  
    int artist_height;  
};
```

Examples of structs

```
struct Album {  
    string title;  
    int year;  
  
    string artist_name;  
    int artist_age;  
    string artist_favorite_food;  
    int artist_height;  
};
```

```
void func() {  
    Album lemonade = {"Lemonade", 2016,  
                    "Beyonce", 41, "Red Lobster", 169};  
  
    Album four = {"4", 2011,  
                "Beyonce", 41, "Red Lobster", 169};  
}
```

Examples of structs

```
struct Album {  
    string title;  
    int year;  
  
    string artist_name;  
    int artist_age;  
    string artist_favorite_food;  
    int artist_height;  
};
```

```
void func() {  
    Album lemonade = {"Lemonade", 2016,  
                    "Beyonce", 41, "Red Lobster", 169};  
    Album four = {"4", 2011,  
                "Beyonce", 41, "Red Lobster", 169};  
}
```

Examples of structs

```
struct Album {
    string title;
    int year;
    Artist artist
};

struct Artist {
    string name;
    int age;
    string favorite_food;
    int height;
};
```

```
void func() {
    Album lemonade = {"Lemonade", 2016,
    "Beyonce", 41, "Red Lobster", 169};
    Album four = {"4", 2011,
    "Beyonce", 41, "Red Lobster", 169};
}
```

Examples of structs

```
struct Album {
    string title;
    int year;
    Artist artist
};

struct Artist {
    string name;
    int age;
    string favorite_food;
    int height;
};
```

```
void func() {
    Artist beyonce = {"Beyonce", 41,
"Red Lobster", 169};
    Album lemonade = {"Lemonade", 2016,
beyonce};
    Album four = {"4", 2011,
beyonce};
}
```

Revisiting Abstractions

ab·strac·tion

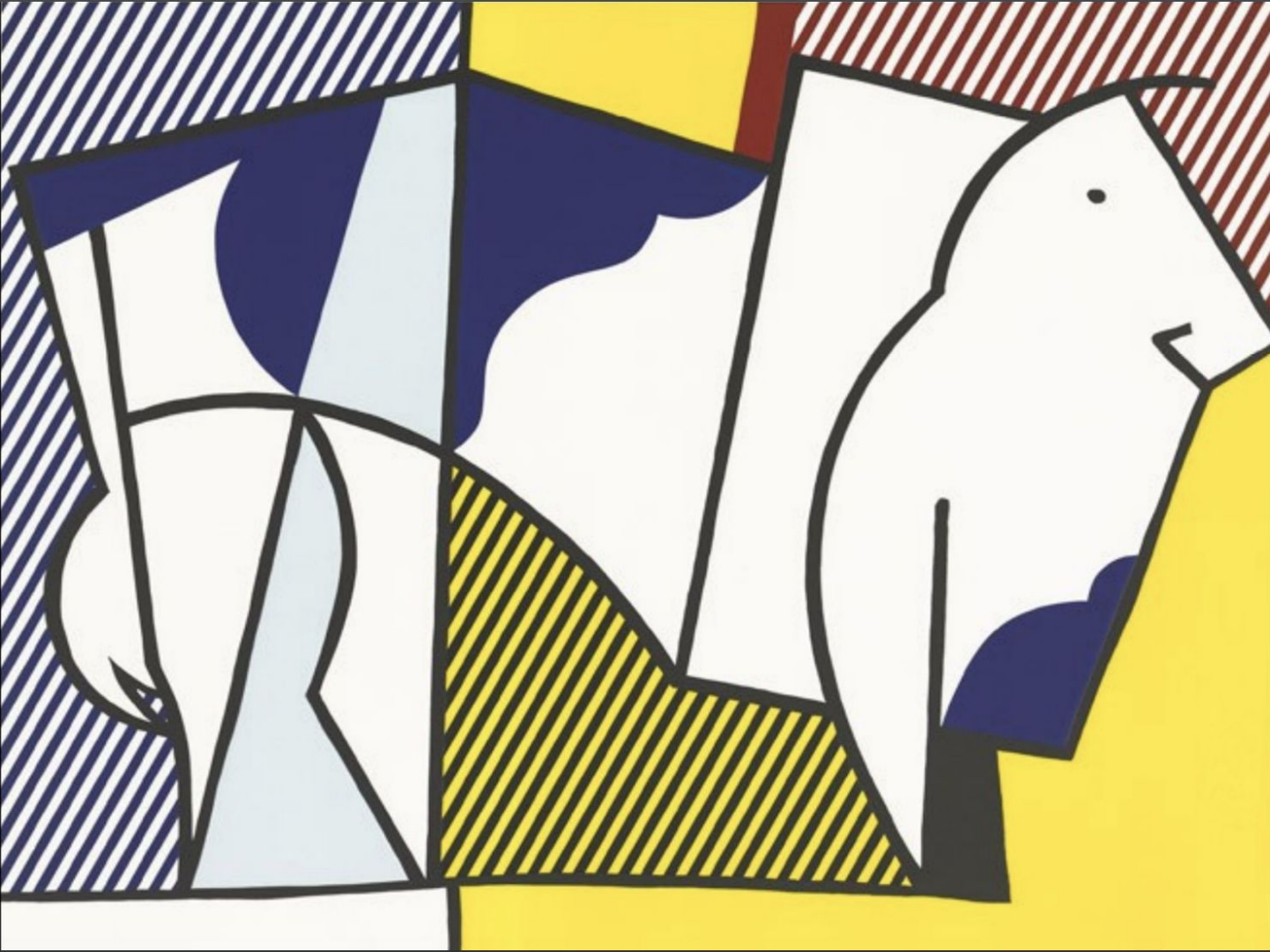
[...]

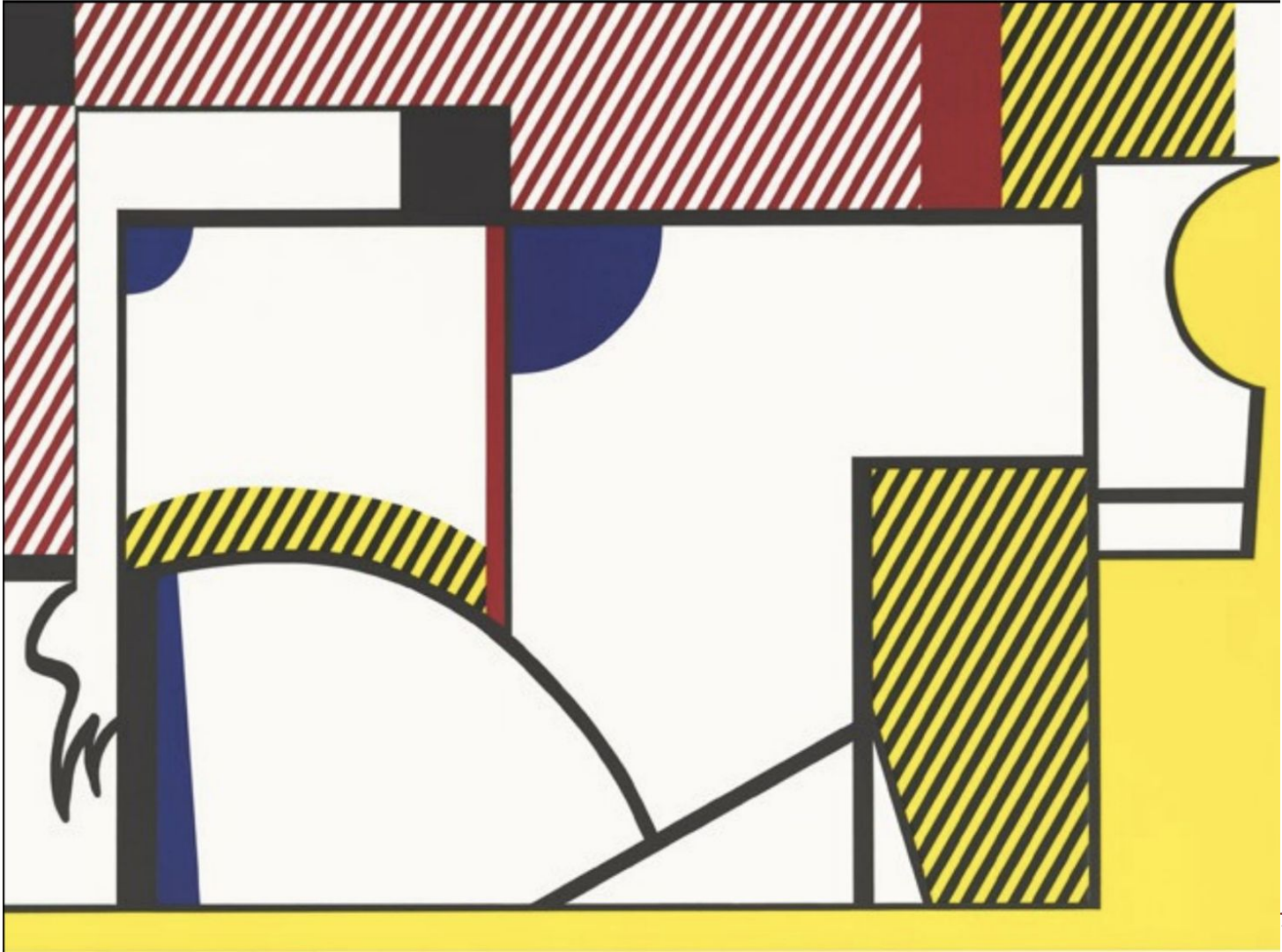
freedom from
representational
qualities in art

Source: Google

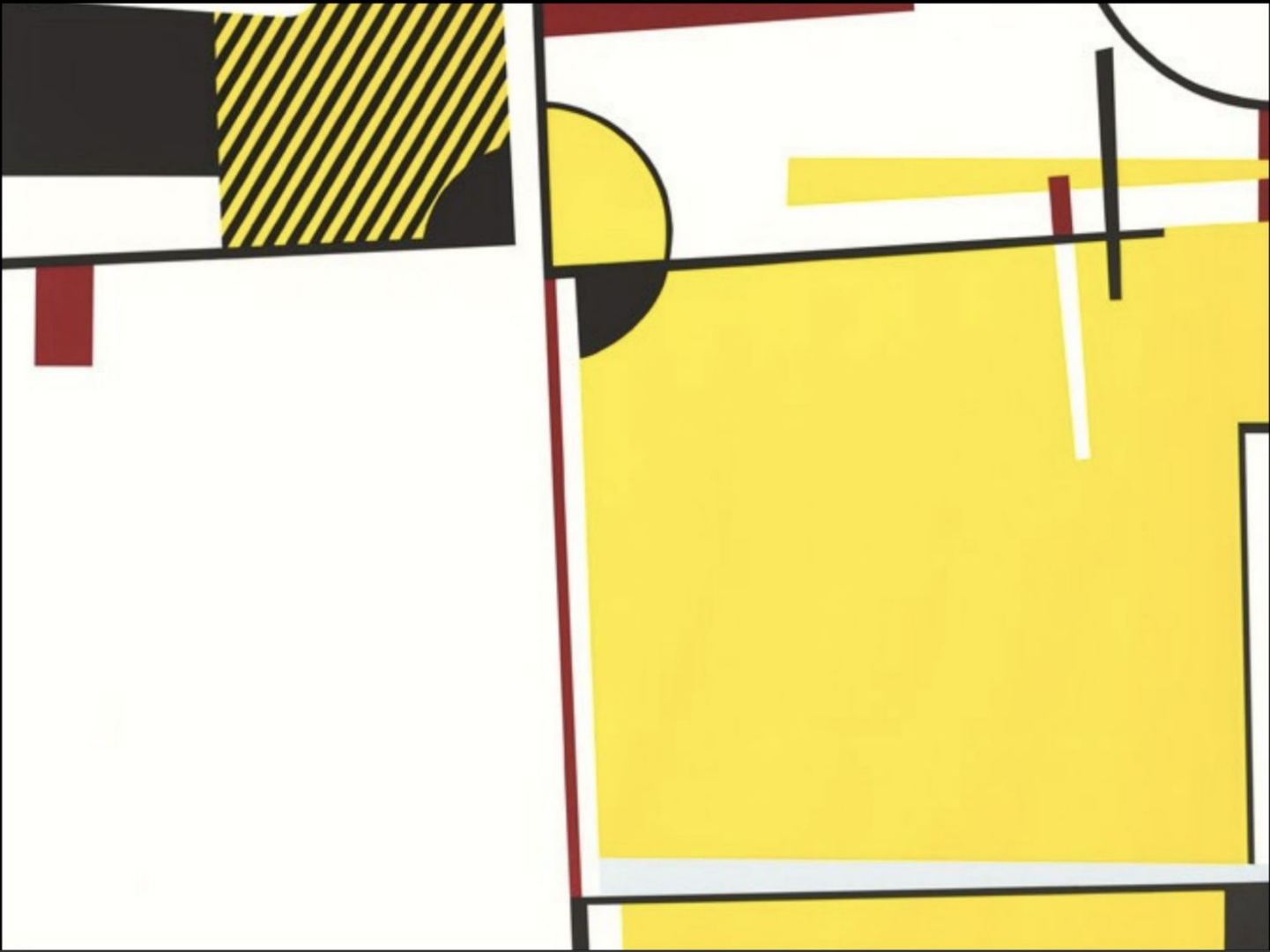












abstraction

Design that hides the details of how something works while still allowing the user to access complex functionality

Classes

Class

- Defines a new data type for our program to use
- Help us create types of objects
 - Which is why we call this object-oriented programming!

Classes

← → ↻ web.stanford.edu/dept/cs_edu/resources/cslib_docs/



The Stanford Libcs106 library, Fall Quarter 2022

Collection classes

Several of the classes represent collections of other objects. These collections work in much the same way as the similar classes in the Standard Template Library (STL).

Each of these collections is documented in its own page, along with sample code. This documentation has been updated for Fall Quarter 2022.

Vector<ValueType>	A Vector is an indexed sequence of values similar to an array .
Grid<ValueType>	A Grid is an indexed, two-dimensional array .
GridLocation	A GridLocation struct is a row/col pair.
GridLocationRange	A GridLocationRange is a two-dimensional range of grid locations.
Stack<ValueType>	A Stack is a linear structure in which values are added and removed only from one end, LIFO .
Queue<ValueType>	A Queue is a linear structure in which values are added at one end and removed from the other, FIFO .
PriorityQueue<ValueType>	A PriorityQueue is a specialized queue in which values are processed in order of priority .
Map<KeyType, ValueType>	A Map maintains an ordered association between keys and values .
HashMap<KeyType, ValueType>	A HashMap is a highly efficient and unordered implementation of the Map abstraction.
Set<ValueType>	A Set is an ordered collection of distinct values .
HashSet<ValueType>	A HashSet is a highly efficient and unordered implementation of the Set abstraction.
Lexicon	A Lexicon is a highly efficient implementation of a word list .

Class

- Defines a **new data type** for our program to use
- Help us create types of objects
 - Which is why we call this object-oriented programming!

Struct

- Way to bundle different types of information
 - Package data into one place
- Like **creating a custom data structure** or variable

GridLocation vs. Grid

```
struct GridLocation {  
    int row;  
    int col;  
};
```

```
class Grid<ValueType>
```

GridLocation vs. Grid

```
struct GridLocation {  
    int row;  
    int col;  
};
```

```
GridLocation chosen = {2, 2};  
cout << chosen.row << endl;  
cout << chosen.col << endl;
```

```
class Grid<ValueType>
```

```
Grid<int> board(3, 3);  
cout << board.numRows() << endl;  
cout << board.numCols() << endl;
```

GridLocation vs. Grid

```
struct GridLocation {  
    int row;  
    int col;  
};
```

```
GridLocation chosen = {2, 2};  
cout << chosen.row << endl;  
cout << chosen.col << endl;
```

```
chosen.row = 3;  
chosen.col = 4;
```

```
class Grid<ValueType>
```

```
Grid<int> board(3, 3);  
cout << board.numRows() << endl;  
cout << board.numCols() << endl;
```

```
board.numRows = 5;  
board.numCols = 4;
```

GridLocation vs. Grid

```
struct GridLocation {  
    int row;  
    int col;  
};
```

```
GridLocation chosen = {2, 2};  
cout << chosen.row << endl;  
cout << chosen.col << endl;
```

```
chosen.row = 3;  
chosen.col = 4;
```

```
class Grid<ValueType>
```

```
Grid<int> board(3, 3);  
cout << board.numRows() << endl;  
cout << board.numCols() << endl;
```

```
board.numRows = 5;  
board.numCols = 4;    board.resize(5, 4);
```

GridLocation vs. Grid

```
struct GridLocation {
    int row;
    int col;
};
```

```
GridLocation
```

```
cout << chosen
```

```
cout << chosen.col << endl;
```

```
chosen.row = 3;
```

```
chosen.col = 4;
```

```
class Grid<ValueType>
```

encapsulation
 process of grouping related information
 and relevant functions into one unit
 and **defining where that information is accessible**

```
endl;
```

```
cout << board.numCols() << endl;
```

```
board.numRows = 5;
```

```
board.numCols = 4;
```

```
board.resize(5, 4);
```


What is a class?

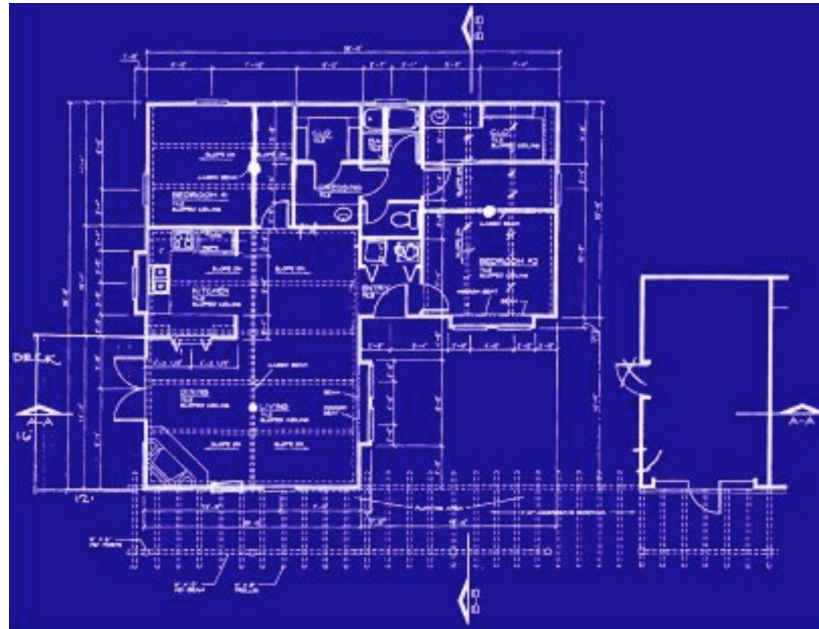
- The only difference between structs and classes are the encapsulation defaults
 - Struct defaults to **public** members (accessible outside the struct itself).
 - Class defaults to **private** members (accessible only inside the class implementation).

What is a class?

- The only difference between structs and classes are the encapsulation defaults
 - Struct defaults to **public** members (accessible outside the struct itself).
 - Class defaults to **private** members (accessible only inside the class implementation).
- Every class has two parts:
 - an **interface** specifying what operations can be performed on instances of the class
 - an **implementation** specifying how those operations are to be performed

Another way to think about classes...

- A blueprint for a new type of C++ **object**!



Another way to think about classes...

- A blueprint for a new type of C++ **object**!
- The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

Another way to think about classes...

- A blueprint for a new type of C++ **object**!
- The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

instance

When we create an object that is our new type, we call this creating an instance of our class.

Another way to think about classes...

- A blueprint for a new type of C++ **object**!
- The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

```
Vector<int> vec;
```

Another way to think about classes...

- A blueprint for a new type of C++ **object**!
- The blueprint describes a general structure, and we can create specific **instances** of our class using this structure.

```
Vector<int> vec;
```

Creates an **instance** of the Vector **class**
(i.e. an object of the type Vector)

Designing C++ Classes

Three Main Parts

- Member variables
 - These are the variables stored within the class
 - Usually not accessible outside the class implementation

Three Main Parts

- Member variables
 - These are the variables stored within the class
 - Usually not accessible outside the class implementation
- Member functions (methods)
 - Functions you can call on the object
 - E.g. `vec.add()`, `vec.size()`, `vec.remove()`, etc.

Three Main Parts

- Member variables
 - These are the variables stored within the class
 - Usually not accessible outside the class implementation
- Member functions (methods)
 - Functions you can call on the object
 - E.g. `vec.add()`, `vec.size()`, `vec.remove()`, etc.
- Constructor
 - Gets called when you create the object
 - Sets the initial state of each new object
 - E.g. `Vector<int> vec;`

How do we design a class?

We must specify the three parts:

- Member variables
 - What subvariables make up this new variable type?
- Member functions
 - What functions can you call on a variable of this type?
- Constructor
 - What happens when you make a new instance of this type?

How would you design a class for...

- 1) A bank account that enables transferring funds between accounts
- 2) A Spotify (or other music platform) playlist

How would you design a class for...

- 1) A bank account that enables transferring funds between accounts
- 2) A Spotify (or other music platform) playlist

We must specify the three parts:

- Member variables: *What subvariables make up this new variable type?*
- Member functions: *What functions can you call on a variable of this type?*
- Constructor: *What happens when you make a new instance of this type?*

Random Bags

Let's write our first class!

Random Bag

- A random bag is a data structure similar to a stack or queue

Random Bag

- A random bag is a data structure similar to a stack or queue
- It supports two operations:
 - add, which puts an element into the random bag, and
 - remove random, which returns and removes a random element from the bag

Random Bag

- A random bag is a data structure similar to a stack or queue
- It supports two operations:
 - add, which puts an element into the random bag, and
 - remove random, which returns and removes a random element from the bag
- Random bags have a number of applications:
 - Simpler: Shuffling a deck of cards.
 - More advanced: Generating artwork, designing mazes, and training self-driving cars to park and change lanes!

Random Bag

- A random bag is a data structure similar to a stack or queue
- It supports two operations:
 - add, which puts an element into the random bag, and
 - remove random, which returns and removes a random element from the bag
- Random bags have a number of applications:
 - Simpler: Shuffling a deck of cards.
 - More advanced: Generating artwork, designing mazes, and training self-driving cars to park and change lanes!

Creating C++ Class



















- Defining a class in C++ (typically) requires two steps:

Creating C++ Class

- Defining a class in C++ (typically) requires two steps:
 - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.

Creating C++ Class

- Defining a class in C++ (typically) requires two steps:
 - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.
 - Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class.

- ▼  **Recursion**
 -  Recursion.pro
 - ▼  Headers
 - >  demo
 -  backtracking.h
 -  hanoi.h
 -  prototypes.h
 -  recursion.h
 - ▼  Sources
 - >  demo
 -  backtrackingwarmup.cpp
 -  boggle.cpp
 -  fundamentalwarmup.cpp
 -  hanoi.cpp
 -  main.cpp
 -  merge.cpp
 -  predict.cpp
 -  sierpinski.cpp

Creating C++ Class

- Defining a class in C++ (typically) requires two steps:
 - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.
 - Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class.
- Clients of the class can then include (using the `#include` directive) the header file to use the class.

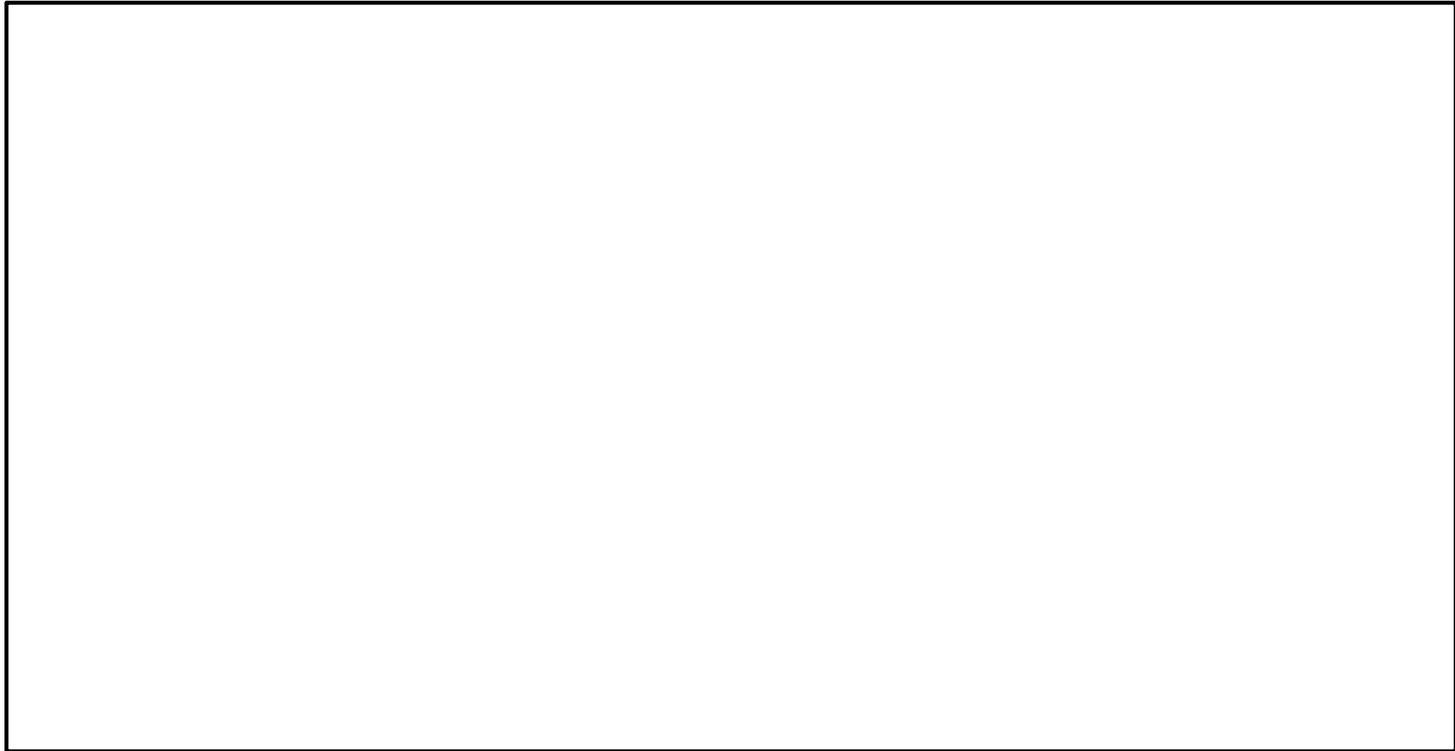
Creating C++ Class

- Defining a class in C++ (typically) requires two steps:
 - Create a **header file** (typically suffixed with `.h`) describing what operations the class can perform and what internal state it needs.
 - Create an **implementation file** (typically suffixed with `.cpp`) that contains the implementation of the class.
- Clients of the class can then include (using the `#include` directive) the header file to use the class.
 - E.g. `#include 'map.h'`, `#include vector.h'`, etc.

Header Files

RandomBag.h

What is in a header file?



What is in a header file?

```
#pragma once
```

This code is called a **preprocessor directive**. It's used to make sure weird things don't happen if you include the same header twice.

What is in a header file?

```
#pragma once
```

```
class RandomBag {
```

```
};
```

This is a **class definition**. We're creating a new class called RandomBag. Like a struct, this defines the name of a new type that we can use in our programs.

When naming classes, use UpperCamelCase.

What is in a header file?

```
#pragma once
```

```
class RandomBag {
```

```
};
```

Don't forget to add the
semicolon!

You'll run into some scary
compiler errors if you leave it out!

What is in a header file?

```
#pragma once
```

```
class RandomBag {
```

```
public:
```

```
private:
```

```
};
```

Interface
(What it looks like)

Implementation
(How it works)

What is in a header file?

```
#pragma once

class RandomBag {
public:

private:

};
```

The **public interface** specifies what functions you can call on objects of this type. (i.e. its methods)

Think things like the Vector
.add() function or the string's
.find().

What is in a header file?

```
#pragma once
```

```
class RandomBag {  
public:
```

```
private:
```

```
};
```

The **public interface** specifies what functions you can call on objects of this type. (i.e. its methods)

Think things like the Vector `.add()` function or the string's `.find()`.

The **private implementation** contains information that objects of this class type will need in order to do their job properly. This is invisible to people using the class.

What is in a header file?

```
#pragma once

class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:

};
```

These are **member functions** of the RandomBag class. They're functions you can call on objects of type RandomBag.

All member functions must be defined in the class definition. We'll implement these functions in the C++ file.

What is in a header file?

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

This is a **member variable** of the class. This tells us how the class is implemented. Internally, we're going to store a `Vector<int>` holding all the elements. The only code that can access or touch this `Vector` is the `RandomBag` implementation

What is in a header file?

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

Implementation Files

RandomBag.cpp

```
#include "RandomBag.h"
```

If we're going to implement the RandomBag type, the .cpp file needs to have the class definition available. All implementation files need to include the relevant headers.

```
#include "RandomBag.h"
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"
```

```
void RandomBag::add(int value){  
    elems.add(value);  
}
```

```
#pragma once  
#include "vector.h"  
class RandomBag {  
public:  
    void add(int value);  
    int removeRandom();  
  
private:  
    Vector<int> elems;  
};
```



```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}
```

The syntax **RandomBag::add** means “the add function defined inside of RandomBag.” The **::** operator is called the **scope resolution operator** in C++ and is used to say where to look for things.

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}
```

If we had written something like this instead, then the compiler would think we were just making a free function named add that has nothing to do with RandomBag's version of add. That's an easy mistake to make!

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}
```

We don't need to specify where `elems` is. The compiler knows that we're inside `RandomBag`, and so it knows that this means "the current `RandomBag`'s collection of elements."

Using the scope resolution operator is like passing in an invisible parameter to the function to indicate what the current instance is.

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();

private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size();
    bool isEmpty();
private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size();
    bool isEmpty();
private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size();
    bool isEmpty();
private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, elems.size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

This code calls our own `size()` function. The class implementation can use the public interface.

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size();
    bool isEmpty();
private:
    Vector<int> elems;
};
```



```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

Let's use it another
place too!

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size();
    bool isEmpty();
private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() {
    return elems.size();
}

bool RandomBag::isEmpty() {
    return size() == 0;
}
```

This use of the `const` keyword means "I promise that this function doesn't change the state of the object."

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size() const;
    bool isEmpty() const;
private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() const {
    return elems.size();
}

bool RandomBag::isEmpty() const {
    return size() == 0;
}
```

We have to remember to add it to the implementation as well!

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size() const;
    bool isEmpty() const;
private:
    Vector<int> elems;
};
```

```
#include "RandomBag.h"

void RandomBag::add(int value){
    elems.add(value);
}

int RandomBag::removeRandom() {
    if (elems.isEmpty()) {
        error("Aaaaahhh!");
    }
    int index = randomInteger(0, size() - 1);
    int result = elems[index];
    elems.remove(index);
    return result;
}

int RandomBag::size() const {
    return elems.size();
}

bool RandomBag::isEmpty() const {
    return size() == 0;
}
```

Note: There are some additional #includes that we'll need. (We'll see them in the actual .cpp file.)

```
#pragma once
#include "vector.h"
class RandomBag {
public:
    void add(int value);
    int removeRandom();
    int size() const;
    bool isEmpty() const;
private:
    Vector<int> elems;
};
```

Using a Custom Class - Code

Takeaways

- Public member variables declared in the header file are automatically accessible in the `.cpp` file.

Takeaways

- Public member variables declared in the header file are automatically accessible in the `.cpp` file.
- As a best practice, member variables should be private, and you can create public member functions to allow users to edit them

Takeaways

- Public member variables declared in the header file are automatically accessible in the `.cpp` file.
- As a best practice, member variables should be private, and you can create public member functions to allow users to edit them
- Member functions have an implicit parameter that allows them to know what instance of the class (i.e. which object) they're operating on

Takeaways

- Public member variables declared in the header file are automatically accessible in the `.cpp` file.
- As a best practice, member variables should be private, and you can create public member functions to allow users to edit them
- Member functions have an implicit parameter that allows them to know what instance of the class (i.e. which object) they're operating on
- When you don't have a constructor, there's a default, zero-argument constructor that instantiates all private member variables
 - (We'll see an explicit constructor next week!)

Structs vs. Classes

Recap

- We can create our own abstractions for defining data types using classes. Classes allow us to encapsulate information in a structured way.
- Classes have three main parts to keep in mind when designing them:
 - Member variables → these are always private
 - Member functions (methods) → these can be private or public
 - Constructor → this is created by default if you don't define one
- Writing classes requires the creation of a header (.h) file for the interface and an implementation (.cpp) file.