Module 13: Recursion

Learning Objectives

1. Design recursive objects to represent a given scenario.
2. Trace the behavior of a recursive function.
3. Design base and recursive cases for a recursion problem.

Process Skills

1. Critical thinking. Using previous knowledge (functions and std::shared_ptrs) to trace recursive functions.
2. Problem-solving. Interpret a programming problem and use recursion to solve it.

Please fill in the roles for each member of your team. Take a look at the description of each role to see its responsibilities. If there are only three people in the group, please assign the same person to the **Presenter** and **Reflector** role. It is a good idea to select roles that you have not recently taken.

Team name: _____ Date: _____

Role	Team Member Name
Manager. Keeps track of time and makes sure everyone contributes appropriately.	
Presenter. Talks to the facilitator and other teams.	
Reflector. Considers how the team could work and learn more effectively.	
Recorder. Records all answers and questions and makes the necessary submission.	

For virtual activities: Once you select your roles, change your Zoom name using the format and example below.

Format: *Group X: First name, Last name initial / Role*

Example: *Group 1: Paul I / Presenter*

Model 1. Recursive objects (27 min) Start time: _____

```
// volunteer.h
#include <iostream>
#include <memory>

class Volunteer {
public:
  Volunteer(const std::string &name,
            std::shared_ptr<Volunteer> next_volunteer)
      : name_(name), next_volunteer_(next_volunteer) { }

  void SetNextVolunteer(std::shared_ptr<Volunteer> next_volunteer) {
    next_volunteer_ = next_volunteer;
  }

  const std::string& Name() {
    return name_;
  }

  void SetHoursWorked(double hours_worked) {
    hours_worked_ = hours_worked;
  }

  double HoursWorked() {
    return hours_worked_;
  }

  double TeamHoursWorked() {
    if (next_volunteer_ == nullptr) {
      return hours_worked_;
    } else {
      return hours_worked_ + next_volunteer_->TeamHoursWorked();
     }
  }

private:
  std::shared_ptr<Volunteer> next_volunteer_;
  std::string name_;
  double hours_worked_;
};
```

```cpp
// main.cc
#include <iostream>
#include <memory>
#include "volunteer.h"

int main() {
 std::shared_ptr<Volunteer> paul
     = std::make_shared<Volunteer>("Paul", nullptr);
 paul->SetHoursWorked(2.0);

 std::shared_ptr<Volunteer> jc
     = std::make_shared<Volunteer>("JC", paul);
 jc->SetHoursWorked(3.0);

 std::shared_ptr<Volunteer> kevin
     = std::make_shared<Volunteer>("Kevin", jc);
 kevin->SetHoursWorked(2.5);

 std::shared_ptr<Volunteer> michael
     = std::make_shared<Volunteer>("Michael", kevin);
 michael->SetHoursWorked(2.5);

 std::shared_ptr<Volunteer> doina
     = std::make_shared<Volunteer>("Doina", michael);
 doina->SetHoursWorked(2.75);

 std::cout << "Team's total hours worked: ";
 std::cout << doina->TeamHoursWorked() << "\n";

 return 0;
}
```

Symbol Table

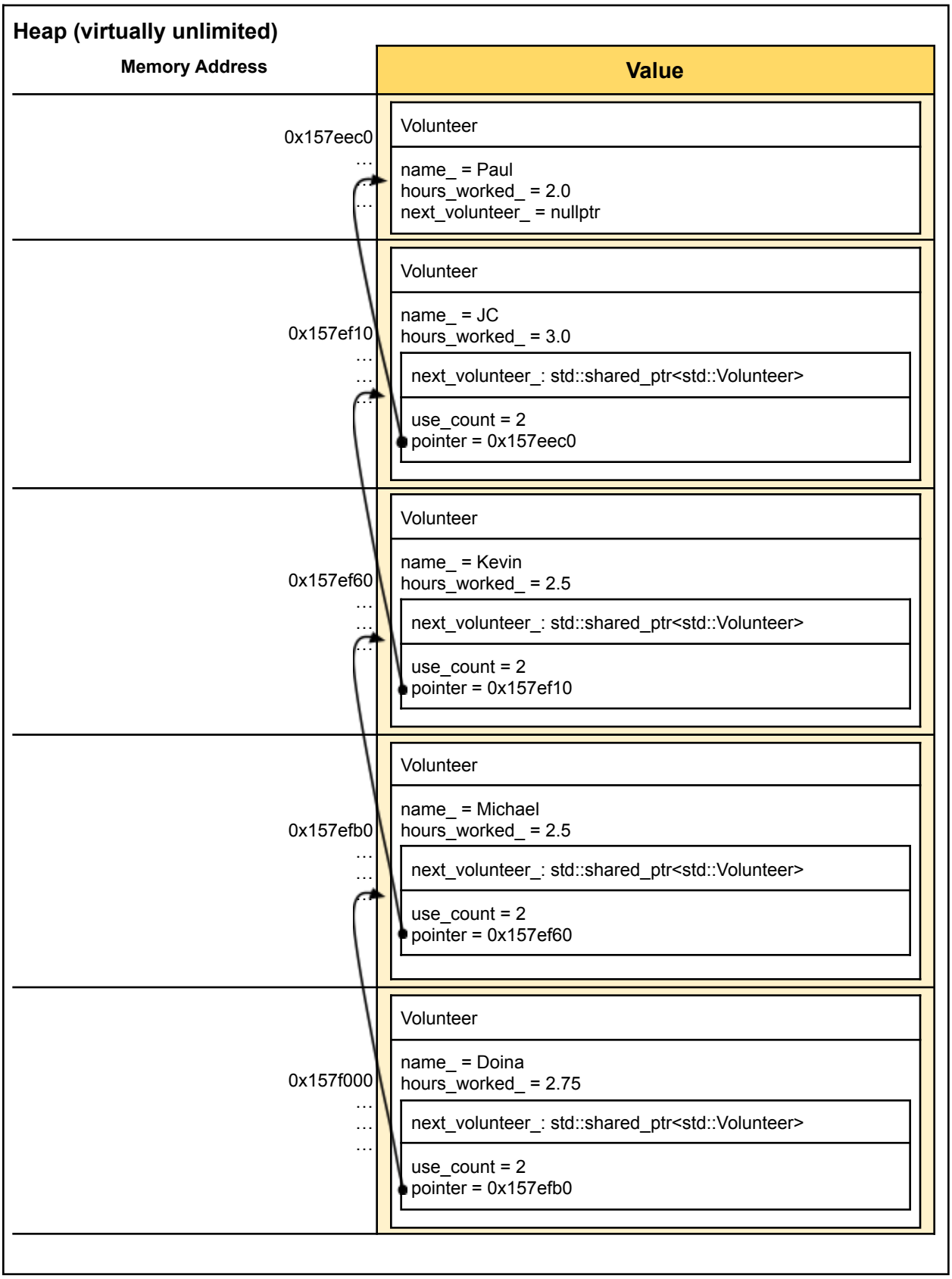
Variable Name	Scope	Type	Memory address*
paul	main()	std::shared_ptr<Volunteer>	0x7ffdb1317040
jc	main()	std::shared_ptr<Volunteer>	0x7ffdb1317030
kevin	main()	std::shared_ptr<Volunteer>	0x7ffdb1317020
michael	main()	std::shared_ptr<Volunteer>	0x7ffdb1317010
doina	main()	std::shared_ptr<Volunteer>	0x7ffdb1317000

** memory addresses are often represented in [hexadecimal format (https://en.wikipedia.org/wiki/Hexadecimal)](https://en.wikipedia.org/wiki/Hexadecimal)*

Memory Visualization before calling `return 0;` in main()

Stack (~8 MB limit)

Memory Address	Value
0x7ffdb1317040 … … …	paul: std::shared_ptr<std::Volunteer> use_count = 2 pointer = 0x157eec0 (see heap visualization)
0x7ffdb1317030 … … …	jc: std::shared_ptr<std::Volunteer> use_count = 2 pointer = 0x157ef10 (see heap visualization)
0x7ffdb1317020 … … …	kevin: std::shared_ptr<std::Volunteer> use_count = 2 pointer = 0x157ef60 (see heap visualization)
0x7ffdb1317010 … … …	michael: std::shared_ptr<std::Volunteer> use_count = 2 pointer = 0x157efb0 (see heap visualization)
0x7ffdb1317000 … … …	doina: std::shared_ptr<std::Volunteer> use_count = 1 pointer = 0x157f000 (see heap visualization)



1. Review the std::shared_ptr object named paul. What is the second argument we pass to its constructor? Place a check (✓) beside your answer.

   a. nullptr ✓
   b. paul
   c. jc
   d. kevin

2. Locate the Volunteer object pointed to by the std::shared_ptr object named paul. What is the value stored in its next_volunteer_ member variable? Place a check (✓) beside your answer.

   a. nullptr ✓
   b. std::shared_ptr<Volunteer> pointing to 0x157eec0
   c. std::shared_ptr<Volunteer> pointing to 0x157ef10
   d. std::shared_ptr<Volunteer> pointing to 0x157ef60

3. What is the address of the Volunteer object pointed to by the std::shared_ptr object named paul. Place a check (✓) beside your answer.

   a. nullptr
   b. 0x157eec0 ✓
   c. 0x157ef10
   d. 0x157ef60

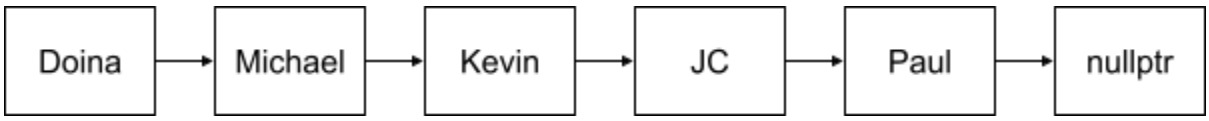
4. Review the std::shared_ptr object named jc. What is the second argument we pass to its constructor? Place a check (✓) beside your answer.

   a. nullptr
   b. paul ✓
   c. jc
   d. kevin

5. Locate the Volunteer object pointed to by the std::shared_ptr object named jc. What is the value stored in its next_volunteer_ member variable? Place a check (✓) beside your answer.

   a. nullptr
   b. std::shared_ptr<Volunteer> pointing to 0x157eec0 ✓
   c. std::shared_ptr<Volunteer> pointing to 0x157ef10
   d. std::shared_ptr<Volunteer> pointing to 0x157ef60

6. Complete the diagram to describe how one Volunteer object links to another Volunteer object through its next_volunteer_ member variable (arrows in the diagram). Just place the value of their name_ variables inside the box. The Volunteer objects we created can form what is called a *Linked List*.



7. Create a Branch class that represents a Food Pantry located at a specific city. Each Branch has a name (the city they are in), and knows the next Branch closest to its location. Represent the next branch as a std::shared_ptr to a Branch. Provide a constructor that accepts a name and a std::shared_ptr to the next branch. No need to provide other member functions (e.g., accessors, mutators).

```
class Branch {
  public:
    Branch(const std::string &name, std::shared_ptr<Branch> next_branch)
        : name_(name), next_branch_(next_branch) { }
  private:
    std::string name_;
    std::shared_ptr<Branch> next_branch_;
};
```

8. Implement the main.cc to create three std::shared_ptrs to Branch objects. The first std::shared_ptr will point to a Branch named Fullerton and does not have a next branch (use nullptr to indicate no next branch). The second std::shared_ptr will point to a Branch named Stanton whose next branch is the Fullerton branch. The third std::shared_ptr will point to a Branch named Garden Grove whose next branch is the Stanton branch.

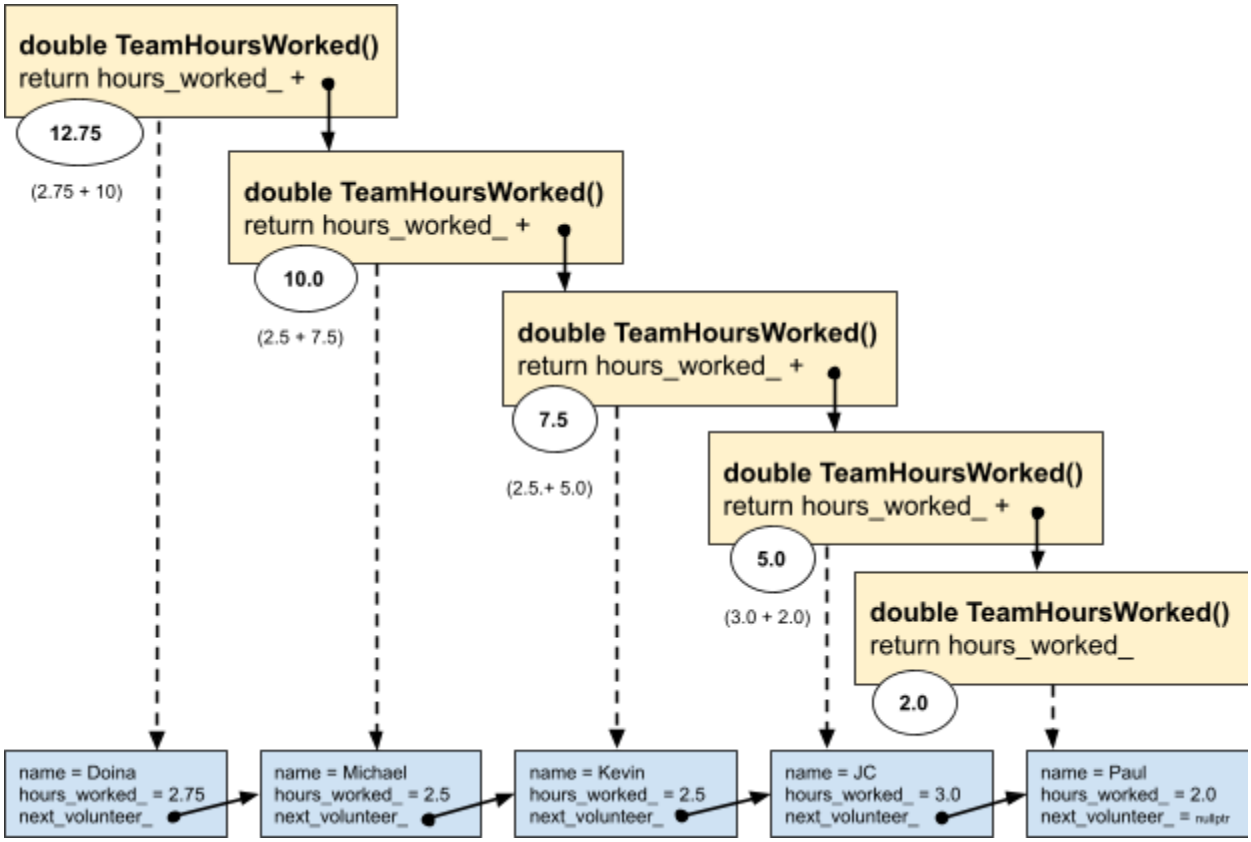
```
int main() {
  std::shared_ptr<Branch> fullerton = std::make_shared<Branch>("Fullerton", nullptr);
  std::shared_ptr<Branch> stanton = std::make_shared<Branch>("Stanton", fullerton);
  std::shared_ptr<Branch> garden_grove = std::make_shared<Branch>("Garden Grove",
                                                                  stanton);
  return 0;
}
```

🛑 **STOP HERE AND WAIT FOR FURTHER INSTRUCTIONS**

Model 2. Recursive processes (24 min) Start time: _____

The visualization below traces the steps performed when doina->TeamHoursWorked() in the main function is called. TeamHoursWorked will compute the total hours worked of the volunteer and all other volunteers in the team (through next_volunteer_).

```
double TeamHoursWorked() {
  if (next_volunteer_ == nullptr) {
    return hours_worked_;
  } else {
    return hours_worked_ + next_volunteer_->TeamHoursWorked();
   }
}
```



9. Consider the Volunteer object pointed to by the std::shared_ptr paul. What is the value of its next_volunteer_ member variable? Place a check (✓) beside your answer.

    a. nullptr ✓
    b. std::shared_ptr<Volunteer> pointing to 0x157eec0
    c. std::shared_ptr<Volunteer> pointing to 0x157ef10
    d. std::shared_ptr<Volunteer> pointing to 0x157ef60

10. If we call the TeamHoursWorked member function on the Volunteer object pointed to by the std::shared_ptr paul, which statement will it execute? Place a check (✓) beside your answer.

```
std::shared_ptr<Volunteer> paul
     = std::make_shared<Volunteer>("Paul", nullptr);
paul->TeamHoursWorked();
```

    a. return hours_worked_; ✓
    b. return hours_worked_ + next_volunteer_->TeamHoursWorked();

> We call the situation wherein a recursive function returns a value or performs an action the *base case* because it does not need the help of any other function to complete its task.

11. If we call the TeamHoursWorked member function on the Volunteer objects pointed to by either the std::shared_ptr jc, kevin, michael, or doina, which statement will they execute? Place a check (✓) beside your answer.

    a. return 0.0;
    b. return hours_worked_ + next_volunteer_->TeamHoursWorked(); ✓

> We call the situation wherein a recursive function requires other functions to return a value or perform an action the *recursive case*. Recursive cases call another recursive function until they reach the *base case*.

12. Consider the Volunteer object pointed to by the std::shared_ptr jc. Calling its TeamHoursWorked member function returns 5.0. How do you think this value was computed? Place a check (✓) beside your answer.

    a. It adds its hours_worked (3.0) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by paul (2.0). ✓
    b. It adds its hours_worked (3.0) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by kevin (2.5)
    c. It adds its hours_worked (3.0) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by michael (2.5).
    d. It adds its hours_worked (3.0) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by doina (2.75).

13. Consider the Volunteer object pointed to by the std::shared_ptr kevin. Calling its TeamHoursWorked member function returns 7.5. How do you think this value was computed? Place a check (✓) beside your answer.

    a. It adds its hours_worked (2.5) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by paul (2.0)
    b. It adds its hours_worked (2.5) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by jc (5.0). ✓
    c. It adds its hours_worked (2.5) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by michael (2.5).
    d. It adds its hours_worked (2.5) to the value returned by the TeamHoursWorked member function of the Volunteer object pointed to by doina (2.75).

14. Consider the Volunteer object pointed to by the std::shared_ptr doina. Calling its TeamHoursWorked member function also calls other objects' TeamHoursWorked member functions. How many TeamHoursWorked member functions are called in total **including** the first call to TeamHousWorked?

5

Recursion Design Process

1. Write your function/member function declaration using the Function Design Process and Function Definition Process.
2. Identify the base case. This is a case where the function's parameters or object's state can give you the answer or perform the task. Write a conditional statement to describe the base case.
3. Identify the recursive case. This is a case where the function's parameters or object's state cannot give you the answer. Write a conditional statement to describe the recursive case. In most situations, the recursive case is the base case's else condition.
4. In the base case, write code to return the answer or perform the task.
5. In the recursive case:
    a. Identify how to produce part of the answer or perform the task using the parameters or object's state.
    b. Identify how another function or object might give you the rest of the answer or perform the rest of the task. If necessary, you can use parameters to delegate the rest of the task to the other function.
    c. If another function will give you the rest of the answer, write code to call that function and combine its result with the answer produced by the current function. If another function will perform the rest of the task, write code to call the function.
6. To use the recursive function, write code to call the function passing in your expected parameters or the member function from an object. Believe in the magic of recursion that will solve the problem or give you the answer.

15. Use the Recursion Design Process to create the TeamMostHours member function for the Volunteer class. The function should use recursion to find the most hours worked by a single team member and return it. In our example, calling TeamMostHours on the Volunteer object for Doina will return 3.0 (JC's hours worked). Just write Volunteer's TeamMostHours member function below.

```
double TeamMostHours() {
  if (next_volunteer_ == nullptr) {
    return hours_worked_;
  } else {
     if (hours_worked_ > next_volunteer_->TeamMostHours()) {
       return hours_worked_;
     } else {
      return next_volunteer_->TeamMostHours();
     }
 }
}
```

🛑 **STOP HERE AND WAIT FOR FURTHER INSTRUCTIONS**

Extra challenge Start time: _____

16. Use the Recursion Design Process to create the TeamAggregateHoursOver member function. It accepts a minimum hours parameter and returns the total hours worked by the team that is over the minimum hours. For example, if the minimum hours parameter is 0.5, and the team members' hours are 1, 2, 0.5, and 3.5, then it will only return 6.5 (1.0 + 2.0 + 3.5).

```
double TeamAggregateHoursOver(double minimum) {
  if (next_volunteer_ == nullptr) {
    if (hours_worked_ > minimum) {
      return hours_worked_;
    }
    else {
      return 0;
     }
  } else {
    if (hours_worked_ > minimum) {
      return hours_worked_ + next_volunteer_->TeamAggregateHoursOver(minimum);
    }
    else {
      return next_volunteer_->TeamAggregateHoursOver(minimum);
     }
  }
}
```

Reflector questions

1. What was the most useful thing your team learned during this session?

2. What made it difficult for you to understand recursion?

3. Did following the recursion design process help you implement recursion? If not, which step or steps were difficult to perform?