Module 14: RAII (Resource Acquisition is Initialization)

Learning Objectives

1. Design and implement destructors.
2. Explain the importance of RAII.

Process Skills

1. Information processing. Extract structural patterns from sample code.
2. Oral and written communication. Describe a technical concept expressed in a written form.

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. Resource Acquisition (15 min) Start time: _____

```
// basketstand.h
class BasketStand {
 public:
  BasketStand() : BasketStand(10) {}
  BasketStand(int total) : total_(total), current_(total) {}
  bool TakeBasket() {
    if (current_ > 0) {
      current_--;
      return true;
    }
    return false;
  }
  bool ReturnBasket() {
    if (current_ < total_) {
      current_++;
      return true;
    }
    return false;
  }
  void DisplayBasketUsage() {
    std::cout << current_ << " of " << total_ << " available.\n";
  }

 private:
  int current_;
  int total_;
};
```

```
// customer.h
#include <iostream>
#include <memory>
#include <vector>
#include "basketstand.h"

class Customer {
public:
 Customer(std::shared_ptr<BasketStand> basket_stand)
     : basket_stand_(basket_stand) {
   basket_stand_->TakeBasket();
 }
 void Take(const std::string &product) {
   products_.push_back(product);
 }
 void CheckOut() {
   for (std::string product : products_) {
     std::cout << product << "\n";
   }
 }
 void ReturnBasket() {
   basket_stand_->ReturnBasket();
 }

private:
 std::vector<std::string> products_;
 std::shared_ptr<BasketStand> basket_stand_;
};
```

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

void SimulateCustomer(std::shared_ptr<BasketStand> basket_stand) {
 Customer paul(basket_stand);
 paul.Take("apple");
 paul.Take("banana");
 paul.CheckOut();
}

int main() {
 std::shared_ptr<BasketStand> basket_stand
     = std::make_shared<BasketStand>();
 basket_stand->DisplayBasketUsage();
 SimulateCustomer(basket_stand);
 basket_stand->DisplayBasketUsage();
 return 0;
}
```

Screen output:
10 of 10 available.
apple
banana
9 of 10 available.

1. What object's member function is called when a Customer object is instantiated? Place a check (✓) beside your answer.

   a. BasketStand::TakeBasket(); ✓
   b. BasketStand::ReturnBasket();
   c. Customer::Take();
   d. Customer::ReturnBasket();

2. What happens to the BasketStand object in the heap when a Customer object is instantiated? Place a check (✓) beside your answer.

   a. If current_ is not 0, it decreases by 1. ✓
   b. If current_ is not equal to the total_, it increases by 1.
   c. If total_ is not 0, it decreases by 1.
   d. If total_ is not equal to the current_, it increases by 1.

3. How many baskets are available after running the SimulateCustomer function?

9

4. The current design requires developers to call the ReturnBasket member function after a Customer checks out. What is the problem if the developer forgets to call the ReturnBasket member function?

If there are more customers than the total number of baskets in the BasketStand, then the store will run out of baskets.

5. What is the problem if the developer calls the ReturnBasket member function multiple times?

There will be a mismatch between the number of baskets in the basket stand and what was actually borrowed. For example, if five customers took a basket and ReturnBasket was called on one customer five times, it would seem that all baskets were returned when other customers were still holding a basket.

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

Model 2. Destructors and RAII (15 min) Start time: _____

```
// customer.h (revised)
#include <iostream>
#include <memory>
#include <vector>

#include "basketstand.h"

class Customer {
 public:
  Customer(std::shared_ptr<BasketStand> basket_stand)
      : basket_stand_(basket_stand) {
    basket_stand_->TakeBasket();
  }

  ~Customer () {
    std::cout << "Destructor called\n";
    basket_stand_->ReturnBasket();
  }

  void Take(const std::string &product) {
    products_.push_back(product);
  }
  void CheckOut() {
    for (std::string product : products_) {
      std::cout << product << "\n";
    }
  }

 private:
  std::vector<std::string> products_;
  std::shared_ptr<BasketStand> basket_stand_;
};
```

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

void SimulateCustomer(std::shared_ptr<BasketStand> basket_stand) {
 Customer paul(basket_stand);
 paul.Take("apple");
 paul.Take("banana");
 paul.CheckOut();
}

int main() {
 std::shared_ptr<BasketStand> basket_stand
     = std::make_shared<BasketStand>();
 basket_stand->DisplayBasketUsage();
 SimulateCustomer(basket_stand);
 basket_stand->DisplayBasketUsage();
 return 0;
}
```

Screen output:
10 of 10 available.
apple
banana
Destructor called
10 of 10 available.

6. The revised code uses a *destructor*. What is the syntax for writing a destructor? Place a check (✓) beside your answer.

   a. <Class name>();
   b. ~<Class name>(); ✓
   c. Destructor();
   d. ~<Class name>([parameter list]);

7. What will happen when the Customer's destructor is called? Place a check (✓) beside your answer.

   a. The BasketStand object's count is set to 0.
   b. The BasketStand object's count is set to total.
   c. The BasketStand object's count decreases by 1 if it is greater than 0.
   d. The BasketStand object's count increases by 1 if it is less than the total. ✓

8. Trace the program and analyze the screen output. When do you think a *destructor* is called? Place a check (✓) beside your answer.

    a. An object's destructor is called when its CheckOut member function is called.
    b. An object's destructor is called when it is instantiated.
    c. An object's destructor is called when its containing function exits, or it is explicitly removed from memory. ✓
    d. An object's destructor is called when there is not enough space in the heap.

9. The term *Resource Acquisition is Initialization* (RAII) is a software design that gives objects the responsibility of acquiring and releasing resources it uses. What do you think are the benefits of using RAII in writing your code? Place a check (✓) beside your answer. Check all that apply.

    a. It avoids cases where a developer might forget to write code that releases acquired resources (e.g., returning the basket in our example). ✓
    b. The logic for acquiring and releasing resources is defined inside the class (proper encapsulation), making it more readable and predictable. ✓
    c. It requires more coding because you need to create a destructor.
    d. It can reduce the amount of memory a program takes up because resources are released as soon as the object goes out of scope. ✓

10. The VolunteerTracker class keeps track of all Volunteers currently working at a food pantry. It provides a StartWork member function that accepts a Volunteer's ID and stores it in the working_volunteers_ vector. The EndWork member function removes a Volunteer's ID from the working_volunteer_ vector. Managers can use this class to track which Volunteers are available to help out in the food pantry.

    Create a Volunteer class that stores the name and ID of a volunteer. Create a constructor that accepts a name, id, and std::shared_ptr to a VolunteerTracker object and assigns them to the appropriate member variable. Additionally, the constructor should call the StartWork member function of the VolunteerTracker object and pass its ID.

    Create a destructor for the Volunteer class that calls the VolunteerTracker object's EndWork member function and passes its ID. As a result, it will remove itself from the list of working volunteers when it is destroyed.

```
// volunteertracker.h
#include <algorithm>
#include <iostream>
#include <vector>

class VolunteerTracker {
public:
 void StartWork(int id) {
   working_volunteers_.push_back(id);
 }




 void EndWork(int id) {
   std::vector<int>::iterator it =
       std::find(working_volunteers_.begin(),
                 working_volunteers_.end(), id);
   if (it != working_volunteers_.end()) {
     working_volunteers_.erase(it);
   }
 }
 void DisplayWorkingVolunteers() {
   if (working_volunteers_.size() > 0) {
     std::cout << "Working volunteer IDs:\n";
     for (int id : working_volunteers_) {
       std::cout << id << "\n";
     }
   } else {
     std::cout << "No working volunteers.\n";
   }
 }

private:
 std::vector<int> working_volunteers_;
};
```

```
// volunteer.h
// TODO: Write your Volunteer class below
class Volunteer {
 public:
  Volunteer(const std::string &name, int id,
            std::shared_ptr<VolunteerTracker> tracker)
      : name_(name), id_(id), tracker_(tracker) {
    tracker_->StartWork(id_);
  }
  ~Volunteer() {
    tracker_->EndWork(id_);
  }
 private:
  std::string name_;
  int id_;
  std::shared_ptr<VolunteerTracker> tracker_;
};
```

```
// main.cc
#include <iostream>
#include <memory>

#include "volunteer.h"

void SimulateVolunteer(std::shared_ptr<VolunteerTracker> tracker) {
 Volunteer paul("paul", 112312, tracker);
 Volunteer jc("jc", 112311, tracker);
 tracker->DisplayWorkingVolunteers();
}

int main() {
 std::shared_ptr<VolunteerTracker> tracker
     = std::make_shared<VolunteerTracker>();
 tracker->DisplayWorkingVolunteers();
 SimulateVolunteer(tracker);
 tracker->DisplayWorkingVolunteers();
 return 0;
}
```

11. Recall how the std::shared_ptr class works. Whenever we create a std::shared_ptr, it instantiates an object in the heap. When a std::shared_ptr is destroyed (i.e., goes out of scope, containing function exits), it looks at the use_count to check if it is 0. If it is greater than 0, the use_count decreases by 1. However, if use_count is 0, it removes the object from the heap to prevent memory leaks.

    According to what we've learned so far, discuss as a group what you think happens inside a std::shared_ptr's destructor. Summarize your discussion in the box below.

The std::shared_ptr's destructor most likely checks the value of use_count. If it is greater than 0 then it decreases the use count by 1. However, if it is equal to 1, then it reduces the use_count to 0 and removes the object it points to in the heap using delete.

Reflector questions

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

2. On a scale of 1 to 5, how comfortable were you in understanding and tracing the code we used today? Write 1 for very comfortable and 5 for very uncomfortable. Explain why you felt this way.

3. On a scale of 1 to 5, how comfortable was your team in writing an explanation for the code and concepts we discussed today? Write 1 for very comfortable and 5 for very uncomfortable. Explain why your team felt this way.