1. Differences Between Pointer Types:

Working through this project, I had to get my head around three types of pointers: *raw pointers*, *dynamic raw pointers*, and *smart pointers*. Each had its place, and I could see how one might be more useful than another, depending on the situation.

- **Raw Pointers (Part A)**:
  In **Part A**, I used *raw pointers* to point to `Task` objects that were created on the stack. It was pretty straightforward, and there was nothing fancy about it. I used these pointers to access task descriptions and mark them as completed. Since no memory was being dynamically allocated, I didn't need to manage memory — the system cleaned up after me when the objects went out of scope. It was simple and worked well, but I didn't get to experience the complexities of memory management, which I'd run into later in the project.

Dynamic Memory with Raw Pointers (Part B):
In **Part B**, I stepped up to using *raw pointers* with dynamic memory (`new`), which meant I had to manually allocate and deallocate memory. I had a lot more control, but with that came more responsibility. If I didn't manage memory properly, I risked **memory leaks**. For example, when I removed a task from the array, I had to shift all the tasks down to fill the gap. Here's the code I used to shift tasks:

```
for (int j = i; j < size - 1; j++) {
    tasks[j] = tasks[j + 1];  // Shifting tasks
}
```

- At this point, I realized how careful you have to be with memory when you're not using smart pointers. If I had forgotten to delete memory, I could have ended up with some serious issues like memory leaks.

Smart Pointers (Part C):
Finally, in **Part C**, I used *smart pointers* (`std::unique_ptr`). This was a game-changer. Instead of manually dealing with memory allocation and deallocation, the smart pointer took care of all that for me. I no longer had to worry about freeing memory because **`std::unique_ptr`** does it automatically when it goes out of scope. Here's how I declared the task array:

```
unique_ptr<Task[]> tasks = make_unique<Task[]>(initialCapacity);
```

- Using `std::unique_ptr` was like a breath of fresh air — it made the code simpler, safer, and way easier to maintain.

---

2. Where and Why `delete` Was Used:

In **Part B**, I had to manually manage memory. I used `new` to allocate memory for an array of tasks and `delete[]` to free it when I was done:

```
Task* tasks = new Task[capacity];  // Dynamically allocating memory
delete[] tasks;  // Manually freeing memory
```

If I hadn't used `delete[]`, I would've had memory leaks, and the program would have gradually eaten up more memory, slowing down or even crashing. It's a lot to think about when working with raw pointers — you have to keep track of when to delete memory, otherwise, you're just wasting resources.

In **Part C**, **`std::unique_ptr`** took care of all the memory management automatically. When the `TaskManager` object goes out of scope, the memory is automatically freed. No need for `delete[]` — it was all handled for me, which was much simpler.

---

3. Shallow vs. Deep Copies:

One of the challenges I faced was ensuring I wasn't making **shallow copies** of tasks. In **Part B**, when I removed a task, I had to shift the remaining tasks in the array to fill the gap. If I wasn't careful, I could accidentally create shallow copies, where multiple pointers would point to the same task. If one pointer modified the task, it would affect others, which isn't ideal.

Here's how I made sure I didn't create shallow copies when shifting tasks:

```
for (int j = i; j < size - 1; j++) {
    tasks[j] = tasks[j + 1];  // Shifting tasks
}
```

With **Part C** and **`std::unique_ptr`**, shallow copies weren't a problem. **`std::unique_ptr`** ensures that each task has unique ownership, which means no two pointers can point to the same memory. It automatically prevents shallow copies, which made things much simpler and more reliable.

---

4. Which Pointer Method is Safest and Why:

After using raw pointers, dynamic memory with raw pointers, and smart pointers, I have to say that **smart pointers** are by far the safest and most convenient. Here's why:

- **Automatic Memory Management**:
  `std::unique_ptr` automatically frees memory when it goes out of scope. This takes a huge weight off your shoulders because you don't need to manually track when to free memory. In **Part C**, this meant I didn't need to worry about `delete[]` anymore — the smart pointer did it for me.

- **No Double Freeing**:
  With raw pointers, there's always the risk of **double freeing** memory. If you call `delete[]` on the same memory more than once, it can cause a crash. Smart pointers prevent this because only one pointer can own the memory at any given time.

- **Cleaner Code**:
  Using smart pointers made the code much cleaner. I didn't need extra lines of code to manage memory, and I didn't have to manually free up memory. The smart pointer automatically takes care of it for me, which makes the code easier to read and maintain.