ProjectOOP: Template Usage Explanation

A Technical Deep Dive into `GameList<T>`

Documentation

December 4, 2025

Abstract

This document provides a comprehensive explanation of how C++ templates are used in the ProjectOOP codebase, with a specific focus on the custom `GameList<T>` template class. It covers the rationale behind using templates, memory management mechanics, and comparisons with standard STL containers.

Contents

1 Overview

This document explains the implementation and usage of C++ templates within the project. The core focus is the `GameList<T>` class, a custom container designed to manage game objects like Obstacles, PowerUps, and Coins efficiently while providing type safety and code reusability.

2 What Are Templates?

2.1 Definition

Templates in C++ are a feature that allows you to write **generic code** that works with **any data type**. Instead of writing separate code for each type, you write one template that the compiler instantiates for each type you use.

2.2 Template Syntax

```
template <typename T>
class MyContainer {
    T* data;
    // ... methods that work with type T
};
```

The `typename T` is a **type parameter**—a placeholder for an actual type that will be specified when the template is used.

2.3 How Templates Work

When you write:

```
MyContainer<int> intContainer;
MyContainer<std::string> stringContainer;
```

The compiler generates **two separate classes** at compile time:

- One that replaces `T` with `int`.
- One that replaces `T` with `std::string`.

3 Why Use Templates in This Project?

3.1 Project Requirement

The project avoids using STL containers like `std::vector`, primarily as an academic exercise to demonstrate:

- Understanding of template mechanics.
- Manual memory management.
- Custom data structure implementation.

3.2 Type Safety

Without templates, alternatives would be unsafe or repetitive:

Option A: Type-specific classes (Code Duplication)

```cpp
class ObstacleList { /* manages Obstacle* */ };
class PowerUpList { /* manages PowerUp* */ };
```

Option B: Void pointers (Unsafe)

```cpp
class GenericList {
    void** data;  // Loses type info! Must cast everywhere.
};
```

Option C: Templates (Chosen Solution)

```cpp
template <typename T>
class GameList {
    T** data;  // Type-safe and reusable
};
```

3.3 Code Reuse

With `GameList<T>`, container logic is written **once** and reused for all game object types, following the DRY (Don't Repeat Yourself) principle.

4 The GameList Template Class

4.1 Complete Implementation

Below is the implementation found in `GameList.h`:

```cpp
template <typename T>
class GameList {
public:
    GameList() : mData(nullptr), mCapacity(0), mCount(0) {}

    ~GameList() {
        clear();
        delete[] mData;
    }

    // Disable copy operations
    GameList(const GameList&) = delete;
    GameList& operator=(const GameList&) = delete;

    void add(std::unique_ptr<T> item) {
        if (mCount >= mCapacity) {
            resize(mCapacity == 0 ? 4 : mCapacity * 2);
        }
        mData[mCount++] = item.release();
    }

    void updateAll(sf::Time dt, float speed) {
        for (size_t i = 0; i < mCount; ++i) {
            mData[i]->update(dt, speed);
        }

        // Remove items marked for deletion
        size_t writeIdx = 0;
        for (size_t readIdx = 0; readIdx < mCount; ++readIdx) {
            if (!mData[readIdx]->isRemovable()) {
                if (writeIdx != readIdx) {
                    mData[writeIdx] = mData[readIdx];
                }
```

```
                writeIdx++;
            }
            else {
                delete mData[readIdx];
            }
        }
        mCount = writeIdx;
    }

    void drawAll(sf::RenderWindow& window) {
        for (size_t i = 0; i < mCount; ++i) {
            mData[i]->draw(window);
        }
    }

    // Iterators for range-based for loops
    T** begin() { return mData; }
    T** end() { return mData + mCount; }

    void clear() {
        for (size_t i = 0; i < mCount; ++i) {
            delete mData[i];
        }
        mCount = 0;
    }

private:
    void resize(size_t newCapacity) {
        T** newData = new T*[newCapacity];
        for (size_t i = 0; i < mCount; ++i) {
            newData[i] = mData[i];
        }
        delete[] mData;
        mData = newData;
        mCapacity = newCapacity;
    }

    T** mData;          // Dynamic array of pointers to T
    size_t mCapacity; // Total allocated space
    size_t mCount;    // Number of elements currently stored
};
```

4.2 Template Method Explanation

Method: `add()` When you call `myList.add(...)`, the `unique_ptr` is moved, ownership is released via `.release()`, and the raw pointer is stored in the internal array.

Method: `updateAll()` This method iterates through the array and calls `update()` on every object. It implies that type `T` **must** have an `update` method (Duck Typing).

5 Template Usage in Practice

5.1 Instantiation in TrackManager

In `TrackManager.h`, the template is instantiated three times:

```
class TrackManager {
    // ...
private:
    GameList<Obstacle> mObstacles;
    GameList<PowerUp> mPowerUps;
```

```
    GameList<Coin> mCoins;
};
```

This generates three distinct classes: one managing `Obstacle*`, one for `PowerUp*`, and one for `Coin*`.

5.2 Adding Objects

```
void TrackManager::spawnObstacle() {
    auto train = std::make_unique<TrainObstacle>(/* params */);

    // Add to the GameList<Obstacle>
    // 1. Ownership moved to add()
    // 2. add() releases ownership to raw pointer inside list
    mObstacles.add(std::move(train));
}
```

6 Memory Management

6.1 Ownership Transfer

The project uses a hybrid ownership model:

1. Objects are created using `std::unique_ptr` (modern C++).
2. Ownership is transferred to `GameList` via `std::move`.
3. `GameList` extracts the raw pointer and assumes responsibility for `delete`.

6.2 Manual Management

Since standard containers are not used, `GameList` handles memory manually:

- **Allocation:** `new T*[capacity]` allocates the array.
- **Deallocation:** The destructor calls `clear()` (to delete objects) and then `delete[] mData` (to delete the array).