```cpp
/* A generic game object class
* Acts as a collection of components
* With methods to add/get components (max components is 32)
* and update/render all of them
*/
#pragma once
#include "Component/Component.h"
#include "Math/Vector2.h"
#include "Global/Event.h"
#include <array>
#include <cassert>
#include <bitset>

constexpr size_t MAX_COMPONENTS = 32;
using ComponentID = std::size_t;
using ComponetBitMask = std::bitset<MAX_COMPONENTS>;
using ComponentArray = std::array<Component*, MAX_COMPONENTS>;

class Transform;

class GameObject
{
private:
    bool b_alive{ true };

    std::vector<std::unique_ptr<Component>> m_component_list;
    ComponetBitMask m_component_bitmask;
    ComponentArray m_component_arr;

public:
    GameObject();
    GameObject(Transform& transform);
    ~GameObject() {};
    void Init() {};
    void Update(float deltaTime);
    void Render();

    bool IsAlive() const { return b_alive; }
    void Deactivate();
    void Activate();

    //set object's transform values
    void SetPosition(const Vector2& position);
    void SetRotationAngle(float rotation);
    void SetScale(const float& scale);
    void SetForwardVector(const Vector2& forward_vector);

    //check if the object has a certain component
    template<typename T>
```

```
50      bool HasComponent() const
51      {
52          return m_component_bitmask.test(GetComponentId<T>());
53      }
54
55      //method to add component to the game object
56      //T is the component type, TArgs is a parameter used to construct the
          component
57      template<typename T, typename... TArgs>
58      T& AddComponent(TArgs&&...m_args)
59      {
60          //make sure the component has not been added before
61          assert(!HasComponent<T>());
62
63          //allocating the component type T on the heap
64          //and forward the passed arguments to its constructor
65          T* component = new T(std::forward<TArgs>(m_args)...);
66
67          //set the object that holds the component
68          component->object = this;
69
70          //use unique pointer so that it manages the obj's lifetime
71          std::unique_ptr<Component> comp_uPtr{ component };
72          m_component_list.emplace_back(std::move(comp_uPtr));
73
74          //add to array and bitmask
75          m_component_bitmask.set(GetComponentId<T>());
76          m_component_arr[GetComponentId<T>()] = component;
77
78          //call component init in which each component will get the other
              component or data it needs
79          component->Init();
80
81          return *component;
82      }
83
84      //method to get the wanted component
85      template<typename T>
86      T& GetComponent()
87      {
88          assert(HasComponent<T>());
89
90          auto component_ptr = m_component_arr[GetComponentId<T>()];
91          return *static_cast<T*>(component_ptr);
92      }
93  };
94
95  inline ComponentID GetUniqueComponentID()
96  {
```

```
 97      static ComponentID lastID{ 0u };
 98      return lastID++;
 99  }
100
101  /* Returns a unique ID the first time it's called with a new component
102  then returns the same ID for the same component */
103  template <class T>
104  inline ComponentID GetComponentId()
105  {
106      static ComponentID typeID = GetUniqueComponentID();
107      return typeID;
108  }
109
110  namespace Object
111  {
112      using Ref_List = std::vector<std::shared_ptr<GameObject>>;
113      using Ref = std::shared_ptr<GameObject>;
114  }
115
```