Masters of Negotiated Studies: Negotiated Project 1

Advanced Programming Systems C++ In Games Development

Christopher Boyce

Student Number: 19016871
Email: b0196871j@student.staffs.ac.uk

Supervisor: Shaun Reeves

Table of Contents

Glossary

UE5: Unreal Engine 5 is a popular game engine created by Epic Games and uses C++ for development.

Data Containers / Containers: These data structures store values and virtual objects; examples include Lists, Arrays, and Maps.

Algorithm: A set of instructions that is followed to perform a task.

STL: Standard Template Library is the original software library that Alexander Stepanov developed for C++.

Key Words

C++, Unreal Engine, Advanced Programming, Games Development, Education, Practical Tutorials

Section 1 : Introduction

1.1 Aims

Investigating and developing a suitable framework for learning purposes incorporates programming fundamentals for a technical game engine.

1.2 Objective

- Identify the uses and solutions for the following: Lists, Arrays, Maps, Non STL Data Structures, Link Lists, Stacks, Queues, Binary Trees, C++ Templates, Lambda Functions, Try-Catch, and Smart Pointers.
- Identify and develop suitable solutions for Unreal Engine Development using C++.
- Detail suitable testing methods for the framework and each potential learning point.
- Highlight suitable learning methods for successful attainment from an audience perspective.

1.3 Background

Today's games industry requires much knowledge as a programmer to perform tasks effectively and efficiently; many hobbies and undergraduates cover the main topics of C++ early in their programming careers and then pursue a game engine of their choosing once the base knowledge is understood. This report will cover the more advanced features of C++, give examples in C++, and then implement them into game engines. Finally, work alongside tasks will be created in Unreal Engine to get hands-on with the concepts and produce a small feature within a game.

The first topic covered is Data Containers, including Lists, Arrays, and Maps, covering the search and sorting algorithms associated with them. The following section will cover non-STL data structures such as link lists, stacks, queues, and binary trees. Finally, C++ Templates, Lambda Functions, Try-Catch, and Smart Pointers will be explored to help improve understanding of writing code. Once these topics are covered, the reader should have a greater understanding of advanced C++ concepts and be able to implement them in their future coding endeavor.

1.4 Project Plan

The project will commence by conducting thorough research on each selected topic. We will begin by examining the integration of each learning topic into C++, encompassing use cases, implementation details, and critical functions essential for users to grasp the feature. Examples of code will be provided for developers to use as a reference. Afterward, our focus will shift to comprehending how these features are incorporated into the Unreal Engine.

Following this initial research phase, an outline of the framework will be developed. This involves formulating a plan for each learning topic and identifying how to create a practical use case within the engine. This plan will then be transformed into a working product, accompanied by a well-documented creation process, allowing the reader to follow along.

Commented [SR1]: "an outline for a framework will be developed."

To conclude, once the research is complete, a series of educational resources, alongside a framework, will be created, designed to aid the reader in their skill development.

Section 2 : Literature Review

2.1 Data Containers

Data containers like Lists, Arrays, and Maps are fundamental parts of storing data elements for use in code. Critical fundamentals must be considered when using these containers, especially when searching and sorting the data within them. In game development, this concept can be related to scoreboards. The user may want to display the players' scores in descending order, showing the highest score at the top. To achieve this, all the scores must be found and ordered accordingly. Additionally, searching through the data can help locate a unique user on the scoreboard and display the scores surrounding them.

2.2 Array

An array is a fixed-size collection of elements of homogeneous data types. Array sizes are declared at compile time and accessed via an index. Arrays are stored in consecutive memory locations, allowing for efficient access. However, due to this reason, their size cannot be changed dynamically (Li & Roberts, 2023).

To search an Array using the Standard Library Function "std::find()," the user needs to provide three parameters. The first parameter is the starting point of the Array, the second parameter is the endpoint of the Array, and the third parameter is the value that needs to be searched for. If the value is found in the Array, the function will assign it to a variable (Grimes & Bancila, 2018).

```
#include <algorithm>    // std::find
int main(){
    int ArrayOfInts[] = { 10, 20, 30, 40 };
    int* OutputInt;
    OutputInt = std::find(ArrayOfInts, ArrayOfInts + 4, 30);
    if (OutputInt!= ArrayOfInts + 1)
        std::cout << "Element found in ArrayOfInts: " << *p << '\n';
    else
        std::cout << "Element not found in ArrayOfInts \n";
}
```

To sort an Array is very similar to using the Standard Library Function "std::sort()." This function takes two or three parameters. The most simplistic is to pass at the start and the end of the Array, and it will sort them depending on their type; for numerical values, it will sort them from smallest to biggest, and for Strings and Chars, it will be lexicographical (Grimes & Bancila, 2018). If a more complex sorting method is needed or a unique sorting approach is required, the user can define the third parameter, a function to specify how they want the data sorted.

```
#include <algorithm>

int main() {
    int array[] = { 1, 2, 9, 1, 5, 6 };

    std::sort(array, array + 6);

    for (int i = 0; i < 6; i++) {
        std::cout << array[i] << " ";
    }
}
```

2.3 Vector

A vector is a dynamically adjustable collection of homogenous data types. It allows for inserting, deleting, adding, and removing elements within the vector. Due to this, vectors handle memory management. For example, adding data to the vector reallocates memory to accommodate more elements, ensuring efficient handling of dynamic resizing.

Vectors use the “push_back(10)” function to add them to the vector. In this demonstration, three integers are inserted into the vector. Accessing specific elements is shown using indexing, such as “intVector[1]”, retrieving the second value in the vector. There is an example of how to iterate through the vector using a For Loop if the user needs to access all the data.

The “pop_back()” function showcases the removal of the final element from the vector, showing data removal capability. Furthermore, the “clear()” function removes all the data from the vector. Finally, the “empty()” function returns a Boolean value if the vector has any elements.

```
#include <vector>

int main() {
    std::vector<int> intVector;

    intVector.push_back(10);
    intVector.push_back(15);
    intVector.push_back(20);

    std::cout << "Element at index 1: " << intVector [1] << std::endl;

    for (int i = 0; i < intVector.size(); ++i) {
        std::cout << intVector [i] << " ";
    }

    intVector.pop_back();

    std::cout << "Vector size: " << intVector.size() << std::endl;

    intVector.clear();
    std::cout << "Is it empty now? " << (intVector.empty() ? "Yes" : "No") << std::endl;

}
```

2.4 Maps

Maps stores a list of pairs, referred to as Key and Value. Maps offer efficient searching using the key value. Maps allow for searching, inserting, and deleting elements using these keys. There are two kinds of maps: a default map sorts the elements based on the value of the keys, and an unordered map stores the elements in the order they are inserted (Dmitrović, 2020). Maps are considered an associated container due to one value being linked to another data element; other examples would include Multi-Maps and Sets (Deitel & Deitel, 2016).

The “map.find()” function utilised a binary search function that looks efficiently through the keys. This is possible due to the sorted nature of the Map; therefore, it can see if the key is larger or smaller than the value being searched for; it can then repeat this process until it finds the key. This is important when looking through a large data set and scales logarithmically with the data set size (Deitel & Deitel, 2016).

“Map.insert()” is a function that will add a new pair to the Map if they follow the parameters defined by the Map. The Map will not add the pair if the inserted key is not unique and will do nothing. “Map.erase()” takes a key and will then remove the pair with the corresponding key from the Map (Deitel & Deitel, 2016). When referring to the pair when isolated, the developer uses the “element.first” to refer to the key and the “element.second” to refer to the value.

```cpp
#include <map>

int main() {
    //Creation of Map
    std::map<int, std::string> map = {{10, "Player1"}, {15, "Player2"}, {20,"Player3"}};

    map.insert({ 25, "Player4" }); //Inserting a New Element

    map.erase(25); //Removing An Element

    auto it = map.find(10); //Finding a Value Through a Key
    std::cout << "Found: " << it->first << " " << it->second << '\n';

    for (std::pair<int, std::string> element : map) { //Looping through a Map
         std::cout << element.first << " " << element.second << '\n';
        }
    map.clear(); //Clears the map
    return 0;
}
```

2.5 Binary Tree

Binary trees are a hierarchical data structure. The data structure enters at the root, and off this node, there are two child nodes. The data is looked at; if this is not the data that is being searched for, it will go to the left branch. This is done in C++ as pointers to the left and right nodes. Binary trees are used for efficient data storage and searching and sorting operations (Aho, et al., 1983).

Before creating the tree, nodes need to be created. This will store the data type on each node and point to the left and right branches. Finally, on the branch, the developer needs to make a constructor to initialize a node with a value. For this example, a binary tree using integers will be created.

```cpp
class TreeNode {
public:
    int data;
    TreeNode* left;
    TreeNode* right;

    TreeNode(int val) : data(val), left(nullptr), right(nullptr) {}
};
```

After making this class, the binary tree class will need to be created. This will first have to store the root node that will be accessed first. There will need to be a constructor for the binary tree that can initialize the tree with no current root node.

```cpp
class BinaryTree {
public:
    TreeNode* root;

    BinaryTree() : root(nullptr) {}
```

Insert

The insert function adds new data to the binary tree while keeping the correct order. To do this, a recursive approach looks for the appropriate location on the tree for data insertion. Initially, it checks that the root node is null and then checks the current node value. If the data is smaller, it navigates to the left child node; if it is larger, it moves to the right. This process repeats until the correct position

for the new data is found, ensuring that the data is placed in the tree in the correct order. (Weiss, 2013)

```
void insert(int value) {
        root = insertRec(root, value);
    }

    TreeNode* insertRec(TreeNode* node, int value) {
        if (node == nullptr) {
            return new TreeNode(value);
        }

        if (value < node->data) {
            node->left = insertRec(node->left, value);
        }
        else if (val > node->data) {
            node->right = inserted(node->right, value);
        }
        return node;
    }
}
```

Search

The search function allows the user to access and see if there is a specific value in the tree. It uses the same recursive checks to navigate the tree. It starts from the root of the tree and checks the current node, and if it will, then if not, the targeted value will compare with the current node value. If it is not the target value, it compares the current node's data and, based on the result, proceeds either to the left or right subtree. It will then repeat this process until finding the target; otherwise, it will return null if the value is not found. (Weiss, 2013)

```
TreeNode* search(int val) {
    return searchRecursive(root, val);
}

TreeNode* searchRecursive(TreeNode* currentNode, int val) {
    if (currentNode == nullptr || currentNode->data == val) {
        return currentNode;
    }

    if (val < currentNode->data) {
        return searchRecursive(currentNode->left, val);
    }
    else {
        return searchRecursive(currentNode->right, val);
    }
}
```

2.6 Templates

Templates are a feature that allows the developer to write generic code to work with different data types. Templates provide a way to define functions, classes, or data structures that can operate on data of various types without having to rewrite the same code for each specific type. Writing generic code is helpful because it allows for code reusability, meaning more maintainability for large code bases; it also allows flexibility in the code when the program knows what data type will be passed in (Vandevoorde & Josuttis, 2017).

In this example, this template will swap any data type's values. The top of the code is how to define a template function. From then on, the developer must use generic variable names as they do not know which data type it will be.

```
template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

2.7 Smart Pointers

Smart pointers in C++ are specialized tools similar to raw pointers pointer but automatically manage memory when the program does not need to access the data. There are three types of smart pointers. The first is Unique Pointers.

Unique pointers have exclusive ownership of an object. This ensures there is only one owner of the

```
std::unique_ptr<int> ptr = std::make_unique<int>(42);
```

object and then it can automatically deallocate the memory when the object is destroyed. Unique pointers are move-only, which means they cannot be copied, and they transfer ownership when moved to a new object. (Lippman, et al., 2012)

Shared pointers allow multiple pointers to share ownership of the allocated memory. The memory is deallocated only when there are no more shared pointers referencing the memory. This feature simplifies the sharing of values between different parts of a program and helps with memory management, reducing the chances of memory leaks. (Lippman, et al., 2012)

```
std::shared_ptr<int> ptr1 = std::make_shared<int>(42);
```

A weak pointer is used with sharded pointers to break potential reference cycles. This is when two objects refer to each other. This can cause memory errors as each one will keep the other from being deallocated. To avoid this, a weak pointer can be used to check if each object is using the memory by running a watcher or checking function to see if the memory is being used. It can also be used to check the existence of an object. (Lippman, et al., 2012)

```
std::shared_ptr<int> shared = std::make_shared<int>(42);
std::weak_ptr<int> weak = shared;

std::shared_ptr<int> shared2 = weak.lock();
if (shared2) {
    //access the object safely
}
else {
    // Object has been deleted
}
```

2.8 Try and Catch and Throw

Using try-and-catch statements in C++ allows the user to run functions that will not always be possible. The try statement allows the code to run in the function, and if it is not possible to complete, it will

```
int main() {
    try {
        int number;
        std::cout << "Enter an integer: ";
        std::cin >> number;

        if (std::cin.fail()) {
            throw std::invalid_argument("Invalid input. Please enter an integer.");
        }

        std::cout << "You entered: " << number << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "An exception occurred: " << e.what() << std::endl;
    }

    return 0;
}
```

use the throw function to go to the catch statement, which can either try a different operation or run an error message. They are helpful because they provide a way of handling runtime errors. This gives the program robustness and allows it to keep running despite an error. This program allows the user to input a number; if they input a number, it will print it into the console. If they do not input a valid number, it will throw the program to a catch, outputting an error message to the console. (Green, et al., 2020)

2.9 Lambda Functions

Lambda functions, or inline functions, express anonymous functions without a function name. They are typically used when a short operation needs to be done. Lambda functions are helpful in C++ because they allow for concise code and easier readability due to the operation being done in the current function. They also improve encapsulation as they have a limited scope, meaning there is no unintended access to variables outside the function.

This lambda function example is used to add together numbers and can be used in the main class of this program. In a lambda function, the lambda nature is indicated by square brackets "[]." In the brackets, input parameters are defined with their types and names, like a regular function. Following this, the arrow operator "->" is used to show the return type from the body of the code. Finally, in curly brackets, "{}" holds the operation that is defined.

```
int main() {

    auto lambda_adding = [](int a, int b) -> int { return a + b;};

    int result = lambda_adding(5, 3);

    std::cout << "Result: " << result << std::endl;

    return 0;
}
```

2.10 Linked Lists

Linked lists are non-STL data structures where node elements are connected through pointers. Each node on the list contains data and a pointer to the next node. This allows for easy traversal between elements in memory without the need for contiguous memory. Linked lists are dynamic, which means that data can be added and removed easily, but they are slower when searching for specific elements because they require traversing the list from the beginning. There are multiple types of linked lists, including singly linked lists, which point to the next object in the list; doubly linked lists, which point to both the previous and next locations in memory; and circular linked lists, where the last node's pointer points back to the start of the list (Carey, et al., 2019).

When making a linked list, the developer must first make a class for the node. This will store the data in the linked list and the pointer to the next value in the chain. It is essential to make a constructor that assigns the value and sets the following value to a null pointer, as it has no value.

```
class Node {
public:
    int data;
    Node* next;

    Node(int val) {
        data = val;
        next = nullptr;
    }
};
```

Once the node has been created, the next step is to construct the Linked List class. To begin, it is crucial to make the head of the linked list. This head serves as the initial link in a chain, with each successive element being added under the last in the chain.

This next element is to create a constructor. This must set the head to be a null pointer, as when constructing the Linked List, there may be no data to insert until later in the program.

The script needs defined functions within the class body that the developer will access to manipulate the Link List. This example shows how to insert data into the class. It first creates a new node with the value provided and performs serval checks. The first to see if the linked list is empty and will set it to the top of the chain; if not, it will traverse down the list by checking the next pointer value, and if it is, it will assign it; this repeats until an empty node is found (Drozdek, 2012).

```
class LinkedList {
public:
    Node* head;

    LinkedList() { head = nullptr; }

    void insert(int value) {
        Node* newNode = new Node(value);
        if (head == nullptr) {
            head = newNode;
        }
        else {
            newNode->next = head;
            head = newNode;
        }
    }
};
```

2.11 Stacks

Stacks are a data structure used in C++ that follow the Last in, First Out principle, meaning the first object on a stack will be the last one removed. Stacks use the “push(x)” function to insert an element on the top of the stack. The “peek()” function is viewing the data on the top of the stack. When accessing data lower, the data above needs to “pop(),” which will remove the top element of the stack to access the data below. This data that is removed is no longer part of the stack. Another essential function is “isEmpty(),” which returns a Boolean value indicating if there is data in the stack. This function helps users check if there is data in the stack before attempting to access a null pointer (McDowell, 2015).

```cpp
#include <stack>

int main() {
    std::stack<int> intStack;

    intStack.push(5);
    intStack.push(10);
    intStack.push(15);

    std::cout << "Top element: " << intStack.peek() << std::endl;

    intStack.pop();

    std::cout << "Is the stack empty? " << (intStack.empty() ? "Yes" : "No") << std::endl;
}
```

2.12 Queues

Queues represent a data structure that operates on the first in, first out principle. This means the first element that is added to the Queue is the first that is removed. The Queue uses different terminology

```cpp
#include <queue>

int main() {
    std::queue<int> intQueue;
    intQueue.push(5); // Enqueueing elements to the queue
    intQueue.push(10);
    intQueue.push(15);

    std::cout << "Front element: " << intQueue.front() << std::endl;

    intQueue.pop(); // Dequeuing an element

    std::cout << "Is the Queue empty? " << (intQueue.empty() ? "Yes" : "No") << std::endl;

    std::cout << "Size of the queue: " << intQueue.size() << std::endl;
}
```

from a stack but the same function names. Enqueue is the act of adding an element to the back of the Queue in C++. This is done using the “push(x)” function. The Dequeue or “pop()” function removes the front of the Queue referred to as the head. “front()” function exposes the front element of the front of the Queue without dequeuing them to be manipulated. Additionally, there are functions to check if the queue “isEmpty()” is similar to stacks and a function to check the “size()” of the Queue (McDowell, 2015).

Section 3 : Testing

Black-box testing is a software testing method that only focuses on the application's functionality. It does not look at the internal code or logic of the software. Testers concentrate on inputs and outputs and do not need to know the system's internal workings. This test plan outlines scenarios, test cases, and expected outcomes based on the software's requirements or specifications (Keploy, 2023 A).

White-box testing is a testing of internal code and functionality. The primary goal of white-box testing is to ensure that internal functions work correctly, including checking that differing code paths and internal data structures are correct. White box testing includes unit testing, integration testing, and system testing (Keploy, 2023 B).

3.1 League Table (Binary Tree)

Blackbox Testing

Scenario	Test Method	Outcome
Data Loads into Table	Open Menu and Visual See table load	Table Loads all 100 inputs
Data Loaded is correct	Compare Loaded Data to Data Table	All Data is correct and link to player name
Search Bar allows for input	Enter Random Letters and Numbers	Search Bar Works
Search bar can find a valid name	Enter Valid Name and wait for response	Gives player output from that correct player input
Search bar can find a valid score	Enter Valid score and wait for a response	Gives player output that is correct from the players input
Return Results text is updated when entered	Enter Valid Name and wait for response	The text updates every input letter until correct (At the Start it says TEXTBLOCK)

Whitebox Testing

Test Name	Function Definition	Expected Outcome	Pass / Fail	Comments
UT_BinaryTreeInsertData	Data from the Data Table is added to the Binary Tree	Binary Tree has all elements inserted	Pass	BP_UnitTesting: Assertion passed (Insert Function Sucess)
UT_BinaryTreeInsertionOrder	Data from the binary Tree has been inserted order via the score function	The binary tree has elements ordered all elements	Pass	BP_UnitTesting: Assertion passed (InsertionOrder Success)
UT_BinarySearchFunction	Binary Search Function uses the algorithm to more effectively search the binary tree	Data is returned	Pass	BP_UnitTesting: Assertion passed (BinarySearch Sucessful)
UT_OrderSearchFunction	In Order Search Function takes more steps of find answer	Data is returned	Pass	BP_UnitTesting: Assertion passed (OrderedSearchFuntion Sucessful)
UT_InOrderTraversal	Returns Binary Tree in order	Array of all data is made in numerical order	Pass	BP_UnitTesting: Assertion passed (InOrderTraversal Success)
UT_ReverseOrderTranversal	Returns Binary Tree in reverse order	Array of all data is made in reverse Numerical order	Pass	BP_UnitTesting: Assertion passed (ReverseOrderTraversal Success)

3.2 Battle Pass (Linked List)

Blackbox Testing

Scenario	Test Method	Outcome
Skins Loads into Boxes	Check all 10 battle pass skins appear	Extra Box appears on the end
Highlight over weapon changes main UI weapon Appearance	Highlight over all 10 weapons and see if it changes	All 10 weapons appeared when cursor hovered over
Equip button changes skin in game	Press the equip button and go into the game repeat with random skins	Each skin can be equipped in game
Equipped weapon is the default on the UI weapon	Equip weapon and then un-hover cursor	Weapon reverts to the equipped item at the time
Menu interactions work with the main menu and back button	Make sure going between pages works and equipped item stay the same	Item is saved and going between pages doesn't change this.

Whitebox Testing

Test Name	**Function Definition**	**Expected Outcome**	**Pass / Fail**	**Comments**
UT_LinkListAddNode	Checks that data has been added to the linked list.	Elements are added to the link list and the start of the list has a valid head.	Pass	BP_UnitTesting: Assertion passed (Insert At First Sucess) & BP_UnitTesting: Assertion passed (Checked Not Null)
UT_LinklistInsertStart	Insert function adds to the head of the link list not the start	New head of the link list should be the inputted value	Pass	BP_UnitTesting: Assertion passed (Add Node Sucess)
UT_LinkListDelAtPos	Deletes value based of position entered	Deleted data from specified node	Pass	BP_UnitTesting: Assertion passed (Delete At Pos Correct)
UT_GetPos	The position is returned to the based of inputted value	The correct position of the input data should be returned	Pass	BP_UnitTesting: Assertion passed (Find Pos Success)

3.2 Settings Menu (Stacks & Templates)

Blackbox Testing

Scenario	Test Method	Outcome
Opening Settings Menu Enters you into new settings menu panel	Press the settings button and see	Transitions to the new page
Pressing corresponding panel opens correct submenu	Press each button and visually check the panel appearing	Each one opens the correct panel
Crosshair Sub Menu works when in gameplay sub menu	Check crosshair settings menu	It opens correctly
Crosshair selection works and saves to in game	Select a crosshair and enter the game to see if it changes	It changes to the corresponding crosshair correctly
Back buttons on all sub menus take you back a singular menu	Open all submenus and close them	All of them close correctly
Graphics and animation play when opened	Open all submenus	Animations don't always play and some of the menus have inconsistencies
Changes in sensitivity save to in game and when coming back to the menu	Change sensitivity and feel the change in game. Come back to the menu and make sure it is the same value	All of the step work correctly and sensitivity is saved in menu and in game.
Back button to main menu exits the settings	Press back and see if it works.	Back functionality is correct

Whitebox Testing

Test Name	Function Definition	Expected Outcome	Pass / Fail	Comments
UT_StackPush	Element of data is pushed onto the stack	Inserted element is added to the stack and check to see if the value is correct	Pass	BP_UnitTesting: Assertion passed (Push Onto the Stack)
UT_StackPop	Element of data is removed from the stack	Inserted element is then removed from the stack	Pass	BP_UnitTesting: Assertion passed (Pop Operation Sucessful)
UT_StackIsEmpty	Returns a bool is the stack is empty or not	Insert data and run the function should return false then remove the elements and run again	Pass	BP_UnitTesting: Assertion passed (Stack Not Empty) & BP_UnitTesting: Assertion passed (Stack is Empty)
UT_StackPeek	The data on the top of the stack is shown	Data inserted on the stack should be accessed through this function	Pass	BP_UnitTesting: Assertion passed (Peek Operation Successful)

3.3 Main Menu (Queues)

Blackbox Testing

Scenario	Test Method	Outcome
When game starts main menu appears	Start the game and visually check	It appears
When "SubMenu" is pressed it brings up the correct "SubMenu" eg. Store, Battlepass and League	Traverse through the store battle pass and League pages	Each menu is accessible
Can return from each submenu to the main menu	Go to a sub menu and return to them	Each menu returns to the main page
Settings loads when pressed	Open the settings menu	Goes to the menu
Returns from the settings menu. Important due to jumping between Stack and Que Method	Press back from the settings menu	Goes to the menu
Animations Play each time from returning from each menu	Watch for animation when returning to the main menu	Animations play

White box testing for this class is unnecessary due to its use of the integrated TQueue Data Structure compared to the Stack, Link List, and Binary Tree. This means that the only functionality testing is done in black box testing.

Section 4 : Analysis of Education Resources

Three topics from the research were picked to make educational tutorials. These topics were Stacks (Appendix A), Binary Trees (Appendix B), and Linked Lists(Appendix C). Using other educational resources online and university-provided tutorials to gather teaching methods, each topic was made into practical applications for programmers.

Each of the tutorials was created with undergraduates as the target audience; therefore, basic C++ knowledge was expected, but knowing there was a wide range of students with varying knowledge of Unreal Engine, it assumed that the reader had not interacted with this software before. Each task was broken down into clear and concise steps, each listed on the front content page for easy understanding. Each project's introduction explains the topic being covered and the use cases and requirements for software that was needed to develop them. The inspiration for this was "Code Academy," (Code Academy , 2022) which uses defined clear topics and breaks down the steps into small, readable paragraphs.

Each task step had keywords in bold to ensure the student visually saw the critical parts of each step. This included name conventions, files and folders, and essential buttons to press. To ensure the student could see all steps, screenshots and videos under each instruction were used to visualize the changes to the project. This gives the reader more context to the action that is being performed. Inspiration for this was the Unreal Engine Documentation (Unreal Engine Docs, 2023), which uses this to separate large chunks of text for the user; this system clearly has a benefit with complex subjects and is used throughout all documentation.

At design points for the menus, the reader was allowed to follow the pre-created tutorial or to make their own version; at this point, advice was given to the user on how to make a personalized menu and topics about menu creation. This allowed the user to create something unique at the end, which they can use in a further project.

All blueprints for each topic were integrated through screenshots and showed the progression steps throughout the project. C++ was used extensively, so each script was broken down into many screenshots. Each screenshot included the line numbers so the reader could refer to them for easy following, and the code had several in-depth comments about the functionality. At the end of an

advanced script was a “Common Issues” section, which included steps for debugging code. Included in this section was a link to GitHub of the complete code for the developer if they were struggling to read or debug the code, allowing them to progress after inspecting it.

At the end of each tutorial, extension tasks were given to readers who wanted to complete more steps in the tutorial. This tutorial consisted of adding to the work that had been completed but something achievable without supervision. A video of the completed task was shown to explain the task in better detail.

In advanced topics such as Binary Trees, graphs and visual aids were created to understand the code flow. This was also used when breaking down all the functions and variables in a script, as there was a high amount.

Each tutorial was made so the user had more options and control of the engine. This was done by design to slowly stop handholding and allow users to make design and coding decisions themselves. This was a critical part of learning the processes; therefore, basics such as creating a script and making menus were not repeated but still referred to in the written actions.

Section 5 : Critical Evaluation/Conclusion

Throughout this report, various advanced programming skills were covered and applied to the development of games using Unreal Engine 5 and C++. The primary objectives are to identify and provide solutions, integrate them into an Unreal Engine project, and outline practical learning approaches for the audience.

Data containers such as binary trees, stacks, and link lists highlighted different solutions for storing data compared to STL-Data containers. Each shows a unique use case and benefits potential improvements to efficiency and code complexity. Stacks stood out in the settings menu as a great solution to avoid code bloat and reusability. Meanwhile, Binary trees showcased superior efficiency when handling large datasets due to the binary search functionality and in-order storage.

Templates used in the settings menu also showed the user the benefits of code flexibility, allowing all data types to be stored in a stack. This understanding is critical to programmers as it allows for high code reuse and is an important topic to cover in an educational C++ report.

Exception handling in Unreal Engine is complex to integrate due to engine constraints but can ensure robustness in error handling and avoid crashes. This topic was important to developers as they may not always use the engine but frameworks where it is easily implementable, and it is a pivotal skill to understand when coding at a high level.

Creating educational resources was pivotal to this project as it bridged the gap between theoretical knowledge and practical understanding. The chosen topics were picked to cover a range of topics and to show the reader the implementation in a modern game engine. The use cases were all based on UI design and popular aspects of the main menu systems. Overall, this was a success in theory, but further data could be used to measure the effectiveness of the resources.

Recommendations / Future Work

To improve upon this work, one of the first steps of picking the topics could be improved by getting a professional opinion on what topics they would like to see undergraduates learn. Interviewing several higher-ups in the industry would give better options than a personal opinion from the limited

resources collected. This step would direct the reader to understand relevant topics and allow progression into the games industry.

When creating the topics in the engine, an external examination from a third party to review the code and create testing would be advisable due to personal bias of how the code should be written, such as coding standards and if all functions have been implemented correctly. This could allow for cleaner code or better functionality, leading to a better educational standard.

Testing of the education piece would be the next step. This could include a sample group and interview of the participants after completion to see if there are gaps in the resource or to hear their review of them. This data could then be collected and used to improve them and future tutorials.

Bibliography

Aho, A. V., Ullman, J. D. & Hopcroft, J. E., 1983. *Data Structures and Algorthims.* s.l.:s.n.

Carey, J., Doshi, S. & Rajan, P., 2019. *C++ Data Structures and Algorithm Design Principles.* 1 ed. s.l.:Packt Publishing .

Code Academy , 2022. *Inheritance.* [Online]
Available at: https://www.codecademy.com/resources/docs/cpp/inheritance
[Accessed 01 01 2024].

Deitel, P. & Deitel, H., 2016. *C++ How to Program.* 10 ed. s.l.:Pearson.

Dmitrović, S., 2020. *Modern C++ for Absolute Beginners: A Friendly Introduction to C++ Programming Language and C++11 to C++20 Standards.* s.l.:Apress.

Drozdek, A., 2012. *Data Structures and Algorithms in C++.* 4 ed. s.l.:Cengage India.

Green, D., Guntheroth, K. & Mitchell, S. R., 2020. *Exception Handling C++.* s.l.:s.n.

Grimes, R. & Bancila, M., 2018. *Modern C++: Efficient and Scalable Application Development.* s.l.:Packt Publishing.

Keploy, 2023. *Black-Box Testing.* [Online]
Available at: https://keploy.io/docs/concepts/reference/glossary/black-box-testing/
[Accessed 31 12 2023].

Keploy, 2023. *White Box Testing.* [Online]
Available at: https://keploy.io/docs/concepts/reference/glossary/white-box-testing/
[Accessed 31 12 2023].

Lippman, S., Lajoie, J. & Moo, B., 2012. *C++ Primer.* 5th ed. s.l.:Addison-Wesley Professional.

Li, Z. Y. G. & Roberts, D. E. W., 2023. *Unreal Engine 5 Game Development with C++ Scripting.* s.l.:s.n.

McDowell, G. L., 2015. *Cracking the Coding Interview.* 6th ed. s.l.:s.n.

Unreal Engine Docs, 2023. *Unreal Engine 5 Migration Guide.* [Online]
Available at: https://docs.unrealengine.com/5.3/en-US/unreal-engine-5-migration-guide/
[Accessed 01 01 2024].

Vandevoorde, D. & Josuttis, N., 2017. *C++ Templates: The Complete Guide.* Second Edition ed. s.l.:Addison-Wesley Professional.

Weiss, M., 2013. *Data Structures and Algorithm Analysis in C++.* Fourth Edition ed. s.l.:Pearson.

Appendices

Appendix 1: Stack Menu Creation (Tutorial 1)

Stacks Menu Creation / UE5.3.2

Contents

Introduction

In this Tutorial, we will be making a Stack Class using the Templates feature to make a settings menu that is dynamic and scalable. Many games now take advantage of being able to pop up new menus as the player clicks through the settings. To do this we need to make a system that controls multiple menu screens simultaneously and displays them.

The Content has been broken down into steps please follow them in order as they build off each other.

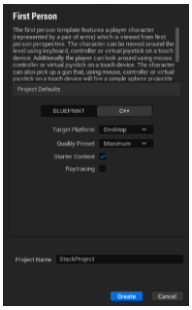
Requirements

This should work on any version of Unreal Engine but for this project it has been tested on **5.3.2**.

Make sure to have **Rider**, Visual Studio, Or an IDE of your choosing installed. In this case, **Rider** is used.

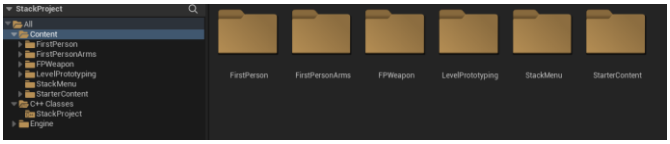
Project Setup

Create a project that uses the **First-Person Template** and uses the **C++.** I added the starter content to get extra assets that will be used for a game loop further down the line. I would recommend doing the same.

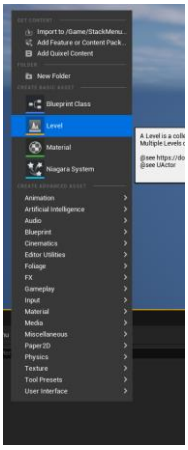


Making Folder and Level Creation

Once the project has been created make a folder in the **Content** Folder called **StackMenu**. This is where we will keep the assets for this task.



In the Folder, Create a new **Level** and call it **MenuScreen**. This will make a new level for the menu to be created on. Once opened, it will be black with no lighting.

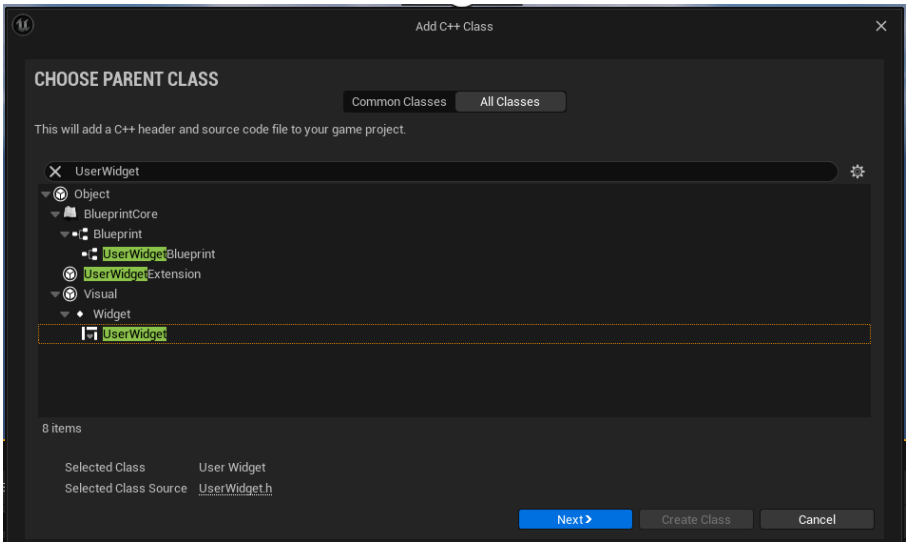


Copy the Lighting folder over the First-Person Map and paste it into the new Level that you have created. This will light up the world and give it a sky box. You may want to edit this to make your menus look different.

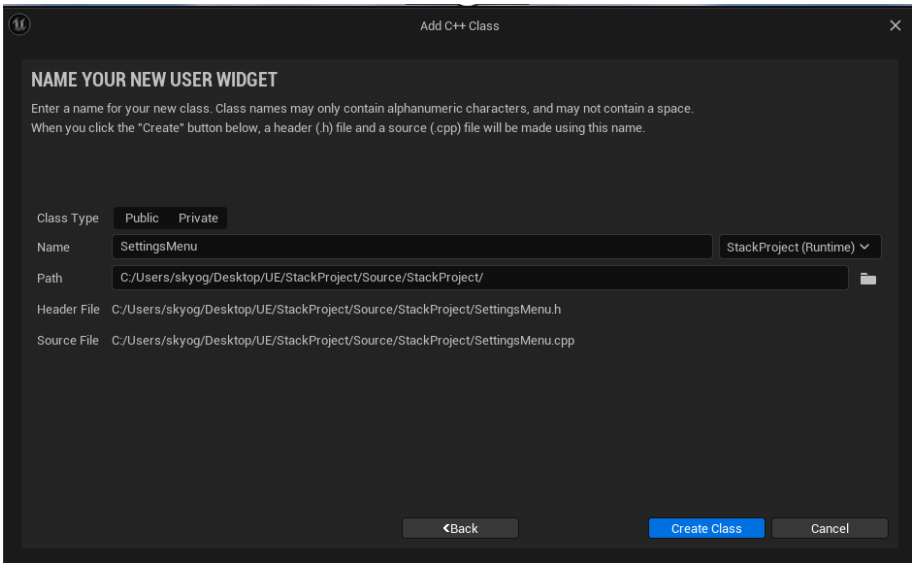
Making the Settings Menu Class and Loading Up Rider

Next, we need to make the main Settings Menu. First, click onto the "**C++ Classes -> Stack Project**" and right-click in the empty space. Press Create a **"New C++ Class".**

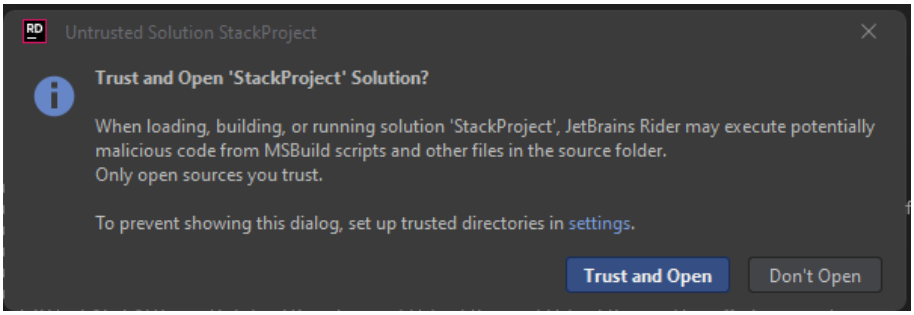
Next Select **All Classes** and Search for the **UserWidget** Class that will be under the **Visuals -> Widget Tab.**



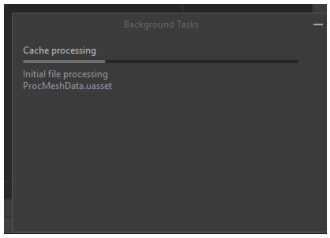
Press **next** and Name the Widget Class as "SettingsMenu" and **Create Class** Button.



As it is the first time Loading Up a C++ Class Rider will Run for the first time make sure to press the **Trust and Open** Button



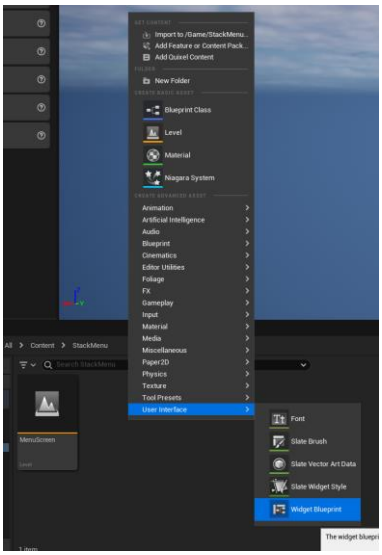
This will open the Project in rider for the first time and will build the solution for you. This will take a few minutes. You can check on the progress by clicking the loading bar at the bottom of the screen. **Wait till this is finished**.



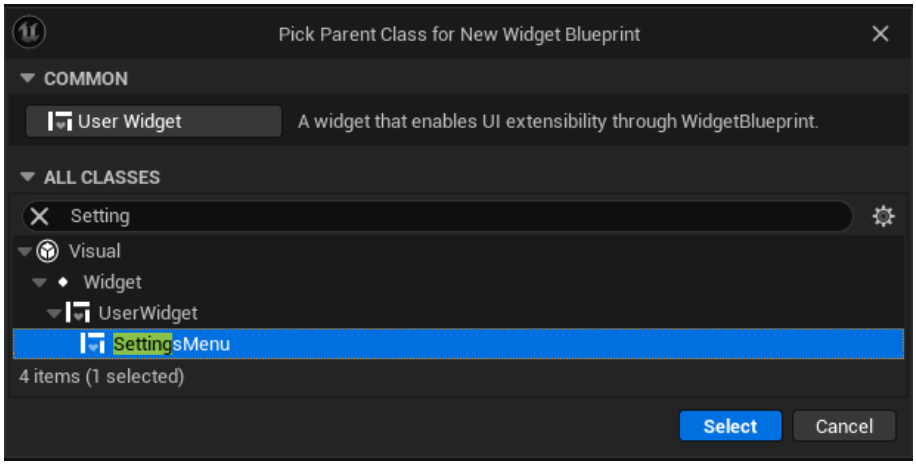
Once this has completed you need to save your Unreal Project by clicking **File -> Save All.** The Closing the Unreal Project.

The In rider in the top right corner you can press the **Run** Button. This will now open the Project using Rider and will allow us to Live Update the code.

Next Step is to create a new Widget Blueprint. Do this by going to **All -> Content -> Stack Menu** and Right Clicking then Hovering over **User Interface** and selecting **Widget Blueprint**.



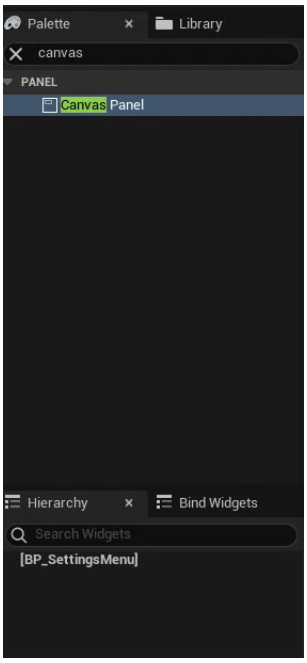
A Box will appear allowing you to select a Parent Class. In the box search for the Class, we just Created Called **SettingsMenu**. Once Selected Rename it to **BP_SettingsMenu**.



You should now need a Widget Blueprint Class that has been Created. Double Click it to open the Editor Window for Widget Class.

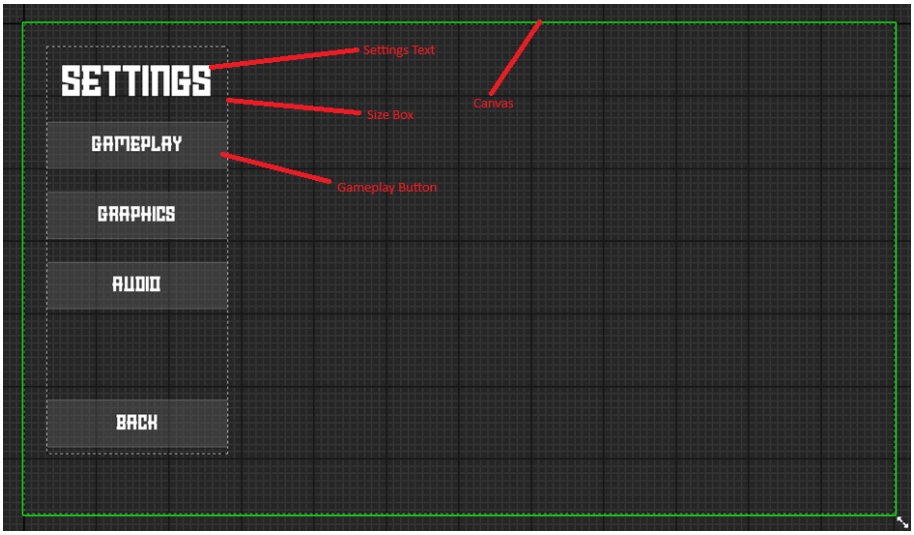
Designing the Menu

First thing to add to the SettingsMenu is a **Canvas** do this by searching in the **Palette** search box found on the left-hand side. And once found drag it onto the Hierarchy which is just below this box. This should then add a dotted box to viewport.



After this you will need to design your Settings Menus. This can be as simple as adding Buttons and a Heading or adding a background. All these options will be found in the Palettes section but **make sure to add 3 Buttons.** One for the **Sound**, one for **Graphics** and one for **Gameplay** as these are what are going to be covered.

Something Nice and Simple Like this will do.



Tips and Tricks for Designing Menus

- Use Size Boxes to get the correct Size Box that you want.
- Try and use Horizontal or Vertical Columns to make sure Buttons Line up.
- You can place text inside the Buttons by Dragging Text into them in the editor.
- Make sure to name all the buttons Correctly to easily find them later.
- Add padding to each button or text to create gaps between each one.
- Import new Fonts into the game to make it look different.
- Change the colours of the buttons to your desired look.

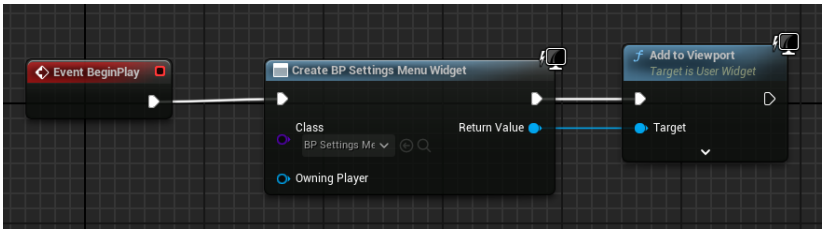
If you want to copy the example above this the Hierarchy.

Once completed make sure to **Save** and **Compile** using the buttons in the Top Left of the Screen.

Adding Menu to the Screen

At the moment when clicking Play nothing is going to happen. This is because the Widget you have created isn't being added to the **Viewport**. To do this you need to find your Level Blueprint. To do this above the window there is **a Blueprints Panel** open this and select **Open Level Blueprints.** This is shown Below.

In the blueprint Window Add the **Create Widget Node** and select the **BP_SettingsMenu** that you have Created. Finally add the **Add to Viewport Node** and Connect to the Pins together.



Now when the game begins it will add the menu to the screen. You will notice that the game is spawning a first-person player. Do change this we will change the game mode. To do this go to **Edit -> Project Settings -> Maps and Modes -> Default GameMode** and change it to **GameMode.**

This will set it to the base class gamemode where a first-person character isn't spawned. You can Change this in the future to support a custom Gamemode if you want to pursue further into the project.

Making The Stack Class

Now we have the Menu first Menu Setup it is time to make the Stack Class. We are using a template type so we can reuse this Stack Class with any Data type. To do this head to the **C++ Classes Folder** and **Create a New C++ Class.** First make sure that is Class doesn't derive from any parent Class by Pressing **None** under the **Common Classes Tab.** And Name it **StackClass**.

It should now open in rider and look like the picture below.

StackClass.h

```
1   // Fill out your copyright notice in the Description page of Project Settings.
2
3   #pragma once
4
5   #include "CoreMinimal.h"
6
7   /**
8    *
9    */
10  class STACKPROJECT_API StackClass
11  {
12  public:
13      StackClass();
14      ~StackClass();
15  };
16
```

The first step you need to do is add the template definer above the Class definition (Line 8) this will tell the code that this is a template Class and will need to be defined.

```cpp
// Fill out your copyright notice in the Description page of Project Settings.

#pragma once

#include "CoreMinimal.h"


template <typename T>
class STACKPROJECT_API StackClass
{
public:
```

The next step is to set up to function Headers in the header File. In this case we are going to add a **Push**, **Pop**, **IsEmpty** and **Peek** function. Each of these functions have different input and return types so make sure they match the code below. Make sure these are in the **Public** section of the header.

Make sure that the **constructor** and **destructor** class have **curly brackets** on the end. **And delete the versions in the CPP Files**.

```cpp
    //Constructor and Destructor
    StackClass(){};
    ~StackClass(){};

    //Push Function -> Adds Element to Stack
    void Push(const T& Element);

    //Pop Function -> Returns Popped Element and Removes it from Stack
    T Pop();

    //IsEmpty Function -> Checks if the Stack is Empty -> Returns Bool
    bool IsEmpty() const;

    //Peek Function -> Checks top element of the Stack -> Returns Element that is Top
    T Peek() const;
```

Finally make a private section of the header file and make a TArray of Type T and name it stack.

```cpp
private:
    //Array that is Used that holds the Information of the Stack
    TArray<T> Stack;
};
```

The next Step is making the Functions for the Stack Class**. In the Header File Under the definitions**, you need to create the functions. This is done in the header file because of the Template Type. The first is the Push Function. Make sure to add the Template Typename as we are using the T inputted. This function simply adds the data to the Stack.

```cpp
//Function Definitions
template <typename T>
void StackClass<T>::Push(const T& Element)
{
    Stack.Add(Element);
}
```

Now define the Pop Function. This function checks if the Stack is empty if it isn't it will look for the Last Element in the stack, remove it and then Return it. If it doesn't exist it will return a default-constructed T object.

```
40  template <typename T>
41  T StackClass<T>::Pop()
42  {
43      if (!IsEmpty())
44      {
45          T PoppedElement = Stack.Last();
46          Stack.Pop();
47          return PoppedElement;
48      }
49      return T();
50  }
```

Now make the Is Empty Function another simple one. Checks if the Stack size is equal to 0.

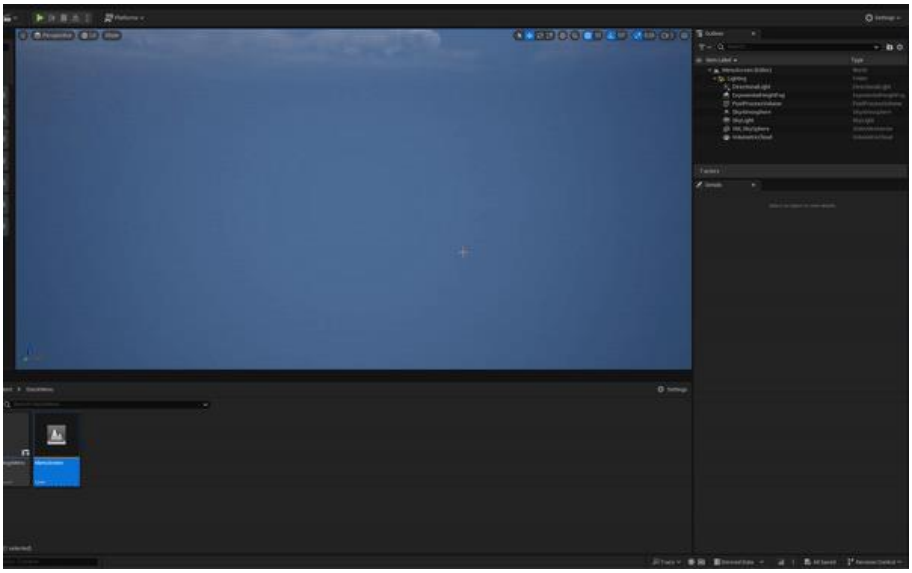
```
52  template <typename T>
53  bool StackClass<T>::IsEmpty() const
54  {
55      return Stack.Num() == 0;
56  }
57
```

And finally, the peek Function. It checks for if it is empty and returns the last item on the Stack. But this time doesn't remove it.

```
58  template <typename T>
59  T StackClass<T>::Peek() const
60  {
61      if (!IsEmpty())
62      {
63          T TopElement = Stack.Last();
64          return TopElement;
65      }
66
67      return nullptr;
68  }
```

If you need to Look at the Full File it is linked here : [AdvancedCPlusPlus/Source/AdvanceCPlusPlus/StackClass.h at main · chris-boyce/AdvancedCPlusPlus (github.com)]

Now you have added all of this save the file and return to the Unreal Engine Editor and Compile the code. This is shown how to do below.



Congratulations if this compiles successfully. You have made a Stack Class. If this doesn't work read over the steps again and follow them to the tee.

Common issues

Make sure that the files are in the public folder.

Make sure to delete the Constructor and Destructor in the CPP.

Make sure that Type Name is above every Function.

Make sure it is all in the Header File.

Have A Look At : [AdvancedCPlusPlus/Source/AdvanceCPlusPlus/StackClass.h at main · chris-boyce/AdvancedCPlusPlus (github.com)]

Using the Stack Class

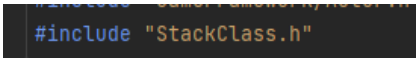
Now we have created the stack class it is time to implement it. To do this **create an C++ Actor Class** Called **SettingsMenuController**.

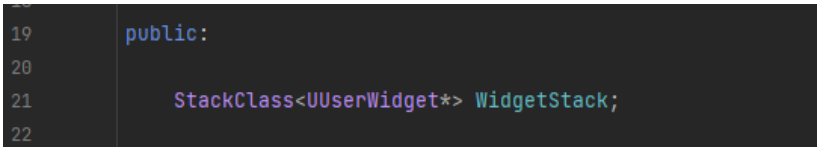
Open the code into rider. In this code you need to add functions called **DisplayTopScreen**, **RemoveTopScreen** and **AddToStack**. As shown below.

```
23      UFUNCTION()
24      void DisplayTopScreen();
25
26      UFUNCTION(BlueprintCallable)
27      void RemoveTopScreen();
28
29      UFUNCTION(BlueprintCallable)
30      void AddToStack(UUserWidget* WidgetToAdd);
```

If you are using rider, right click these functions and generate the **Function Definitions** and it will create the function definitions in the CPP file for you. **Do this for all 3 Functions**. Make sure to added **BlueprintCallable** to the two functions as these will be called from there later.

We also want to make a StackClass that will store are UUserWidgets. Make sure that you include your StackClass.h in the header file at the top of the screen and UUserWidget* is a pointer. Rider will tell you if you do not do this step.





Finally, you need to make 3 variables of UUserWidget* that store the menus that will appear after a button is clicked. Make sure they are EditAnywhere so we can change them in the inspector and Blueprint Read Only so they cant be changed.

Moving onto the CPP of **SettingsMenuController** you should have **the 3 function definitions** as well as the BeginPlay, Constructor and Tick functions.

In the **DisplayTopScreen** Function you need the following code. This will look at the stack using the peek function that we created. It will then add it to the viewport if it is valid.

```cpp
void AMainMenuController::DisplayTopScreen()
{
    UUserWidget* TopWidget = WidgetStack.Peek();
    if(TopWidget)
    {
        TopWidget->AddToViewport();
    }
}
```

The next step is Removing the Top Screen. In this function we will get the top widget and remove it the pop it from the stack.

```cpp
void AMainMenuController::RemoveTopScreen()
{
    UUserWidget* TopWidget = WidgetStack.Peek();
    if(TopWidget)
    {
        TopWidget->RemoveFromViewport();
    }
    WidgetStack.Pop();
}
```

And finally, in the AddToStack Function you need to push the widget that you pass in onto the stack and Display it to the top of the screen.

```cpp
void AMainMenuController::AddToStack(UUserWidget* WidgetToAdd)
{
    WidgetStack.Push(WidgetToAdd);
    DisplayTopScreen();
}
```

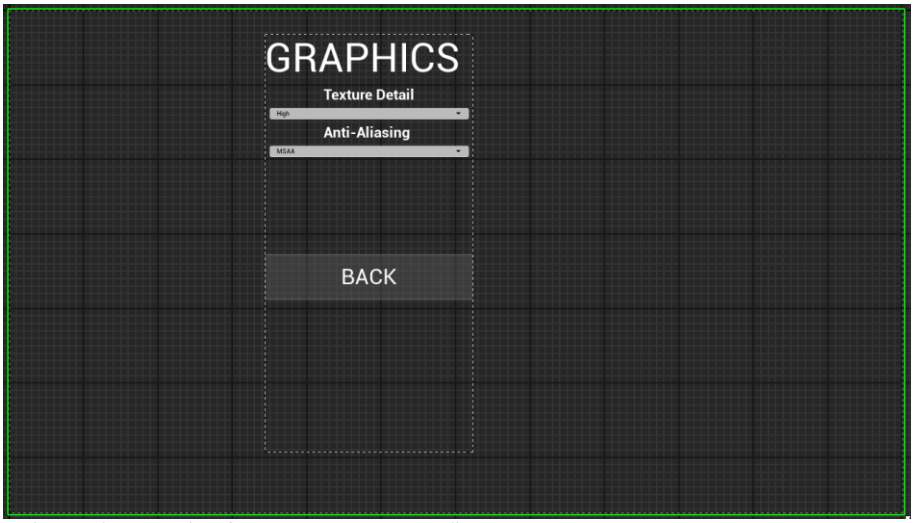
Look at what the script is meant to look like here:

Header : AdvancedCPlusPlus/Source/AdvanceCPlusPlus/SettingsMenuController.h at main · chris-boyce/AdvancedCPlusPlus (github.com)

CPP : AdvancedCPlusPlus/Source/AdvanceCPlusPlus/SettingsMenuController.cpp at main · chris-boyce/AdvancedCPlusPlus (github.com)

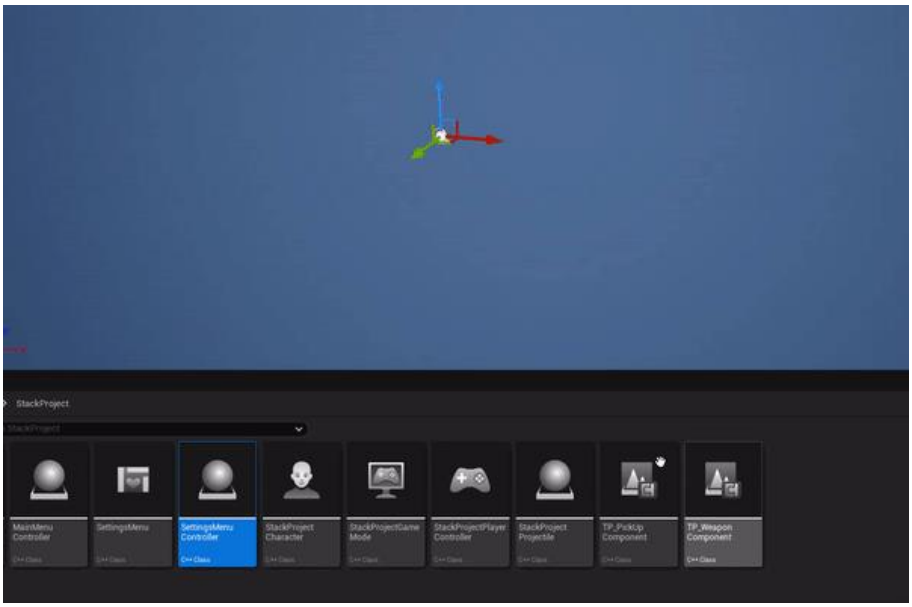
More Menus

This is all the code need for this menu system to work. The next step is to the next set of menus and calling the functions. To do this create the Gameplay, Graphics and Audio Menus. Like we did in Settings Menu. Make sure to derive the Class from UserWidget. See Example Below for Ideas.



Making a Blueprint Class from SettingMenuController

When you have made your **menus create a blueprint derived class of SettingsMenuController**. You can do this by right clicking the C++ in the editor and selecting the option. Place this into your StackMenu Folder. Make sure to name it BP_SettingsMenuController as it important to keep blueprints separate but also know what class the derive from. **See Below.**

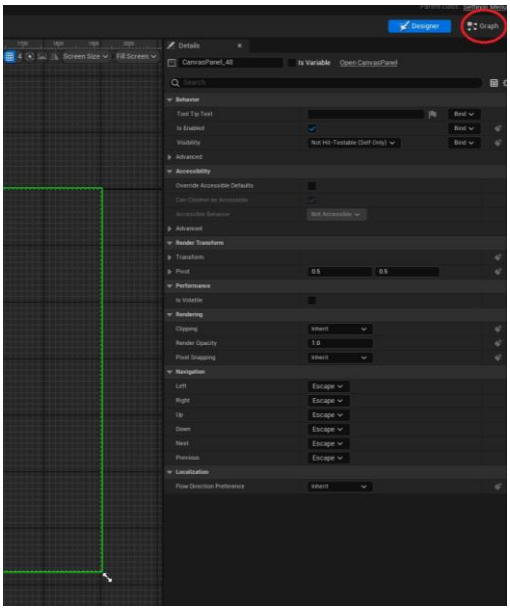


Place this Blueprint Class into the World.

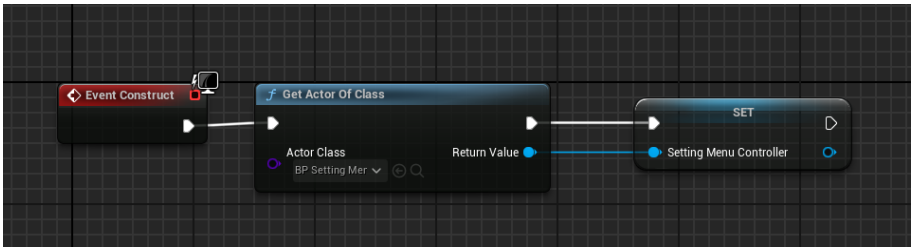
After this **set the variables** we made such as Audio, Gameplay and Graphics menus into the variables that we made. This will be under the **detail tab** when you select the BP_SettingsMenuController in the world. **Make sure these are set or it will crash.**

Setting Up the Blueprint

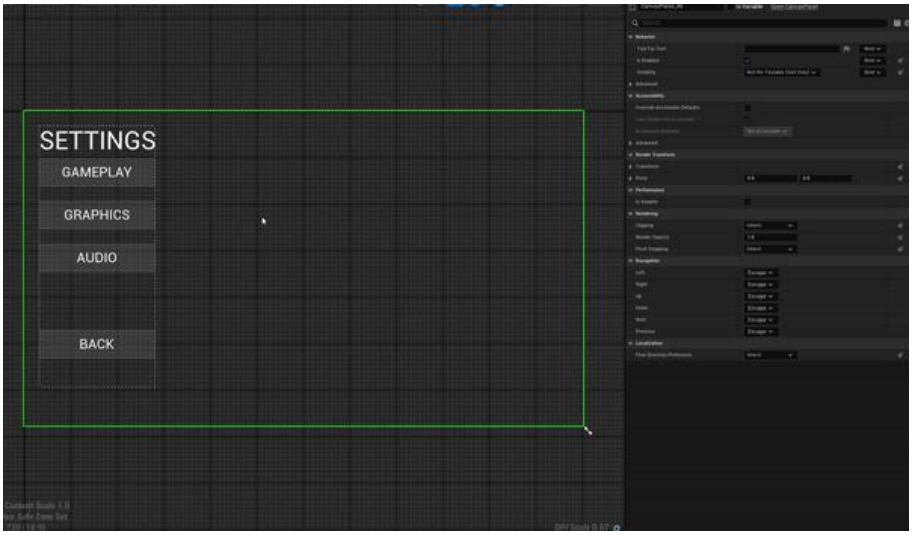
Open the BP_SettingsMenu. It will likely display the menu you have made. Head over to the graph section of the editor located in the top right corner.



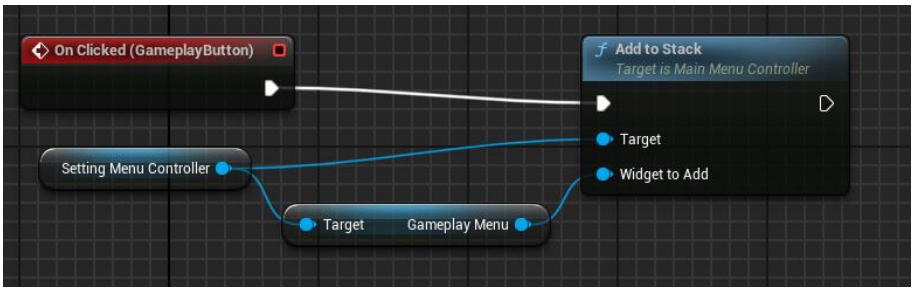
In the constructor event, add the node **Get Actor Of Class** and set Actor Class to your BP_SettingsMenuController. The from the exit node **Promote it to a variable** and call it Settings Menu Controller. This gives you access to the Actor that we placed in the world.



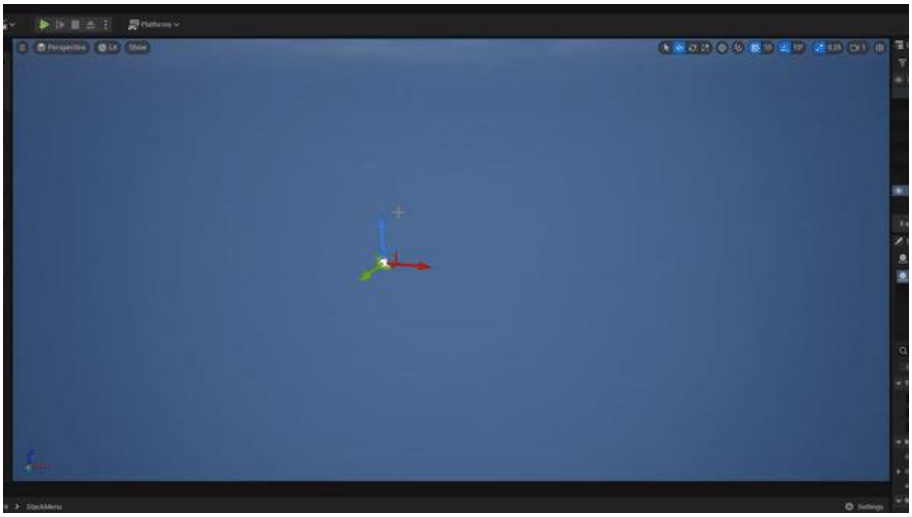
Now create a function from when the button is pressed, you can do this by selecting the button and pressing the create **OnClick** function shown below.



Now we need to call the function on the actor we created to do this. Get the Settings Menu Controller and call the Add to Stack function we created. Also, from the Settings Menu Controller add the corresponding menu and pass it in. Do this for all 3 submenus you have made.



Now when we click the button it should bring up the menu you selected.

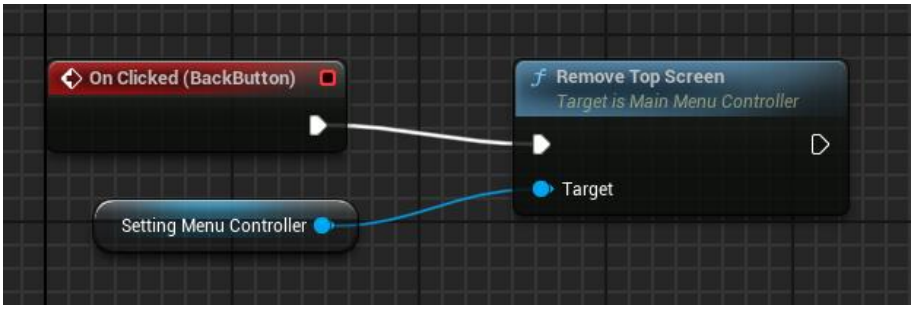


Back Button Functionality

At the moment no functionality has been given to the back button therefore you cannot remove the submenus once you have enabled them.

To do this you need to **recreate the constructor blueprint** like we did before in the BP_GraphicsMenu, BP_AudioMenu and BP_GameplayMenu that you should have made.

After this create an **On Click Event** on the **back button** and run the function **Remove Top Screen.**



Now you should have a functional menu that can pop new menus up and remove the menus when back is pressed. **Congratulations**

Extension Task

Can you now try and make a menu system that can go multi levels as shown below. Try adding a 3rd layer to the menu system? You should have all the code needed just a small bit of blueprint necessary.


SETTINGS
GAMEPLAY
GRAPHICS
AUDIO
BACK

Appendix 2: Binary Tree Creation (Tutorial 2)

Binary Tree Scoreboard / UE5.3.2

Contents

Introduction

This tutorial will continue looking at advanced C++ programming and creating them in Unreal Engine. In this tutorial, we will make a **binary tree** that stores player's names and scores. There will implement a **search function** so people can find their friends and their current score. This will be using **binary search** and **in-order search functions** to see the benefits of each one.

This content assumes you have **completed** the **Stack Menu Creation Tutorial,** not because it will be using any of the code but the fundamentals of using the Widget Creation Tools and Rider as this will not be covered again.

Requirements

This should work on any version of Unreal Engine but for this project it has been tested on **5.3.2**.

Make sure to have **Rider**, Visual Studio, Or an IDE of your choosing installed. In this case **Rider** is used.

Also to make the CSV file Excel is needed; any version will work.

Data Tables and Structs

To store the data that we are going to display we are going to use Unreal Engine **Data Tables**. These are a data store that use **CSV files**. To do this first we need to make a **Struct** that is going to hold these. First make a new C++ Class that uses a **UObject** as it parent class and name it **BinaryTreeNode**.

In the Header File **Above the UCLASS()** definition you need to make a **USTRUCT().** This struct needs to inherit off **FTableRowBase.**

```
10
11      //Data Table Struct
12      USTRUCT(BlueprintType)
13      struct FScoreboardData : public FTableRowBase
14      {
15          GENERATED_BODY()
16      public:
17          UPROPERTY(EditAnywhere, BlueprintReadOnly)
18          FString Name;
19
20          UPROPERTY(EditAnywhere, BlueprintReadOnly)
21          int32 Score;
22      };
```

Make sure to **#include Engine/DataTable.h**

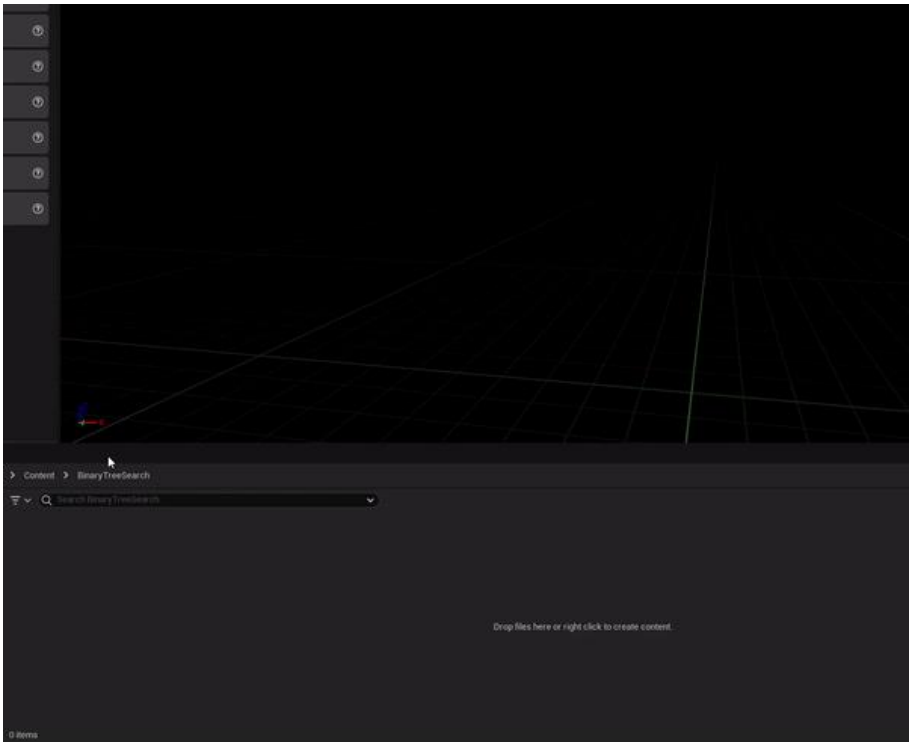
```
#pragma once

#include "CoreMinimal.h"
#include "Engine/DataTable.h"
#include "UObject/NoExportTypes.h"
#include "BinaryTreeNode.generated.h"
```

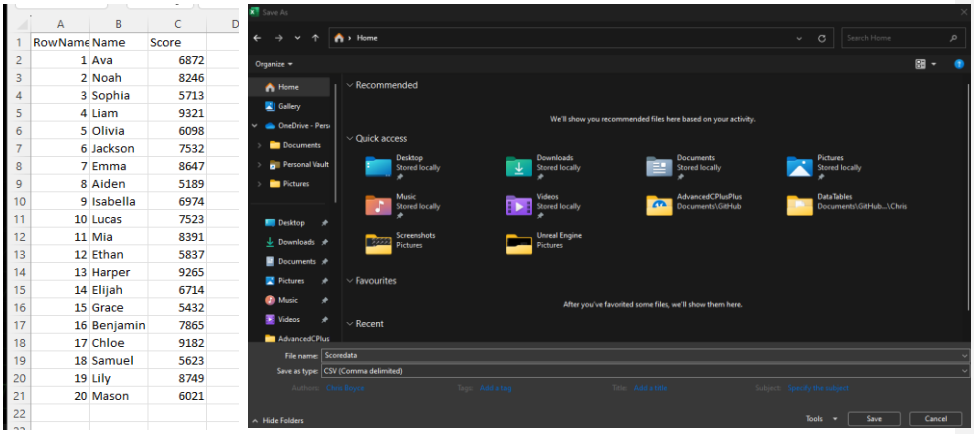
The Struct needs to store an **FString** that is going to store the **Name** and an Int that will hold the score. Make sure they are have the **UPROPERY** EditAnywhere and BlueprintReadOnly.

Make sure to Save and Compile.

Now you need to make a **Data Table**. It is under the **Miscellaneous** Section in Unreal Engine. Make sure to put it in a folder that is easy to find and to **name it appropriately**. This is shown below.

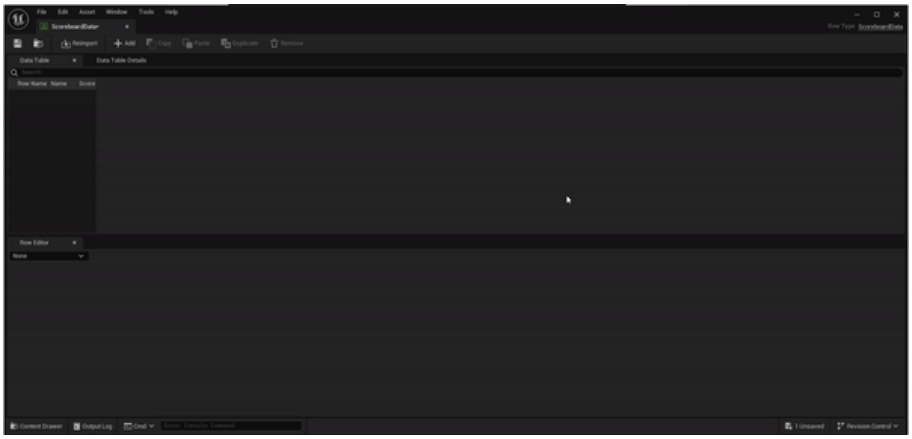


Next you need to make an **Excel File** that is going to hold the data. Add the Titles RowName, Name and Score. Add the data you want to store. Finally Save the document as an **CSV file** and place it somewhere easily accessible.



	A	B	C	D
1	RowName	Name	Score	
2	1	Ava	6872	
3	2	Noah	8246	
4	3	Sophia	5713	
5	4	Liam	9321	
6	5	Olivia	6098	
7	6	Jackson	7532	
8	7	Emma	8647	
9	8	Aiden	5189	
10	9	Isabella	6974	
11	10	Lucas	7523	
12	11	Mia	8391	
13	12	Ethan	5837	
14	13	Harper	9265	
15	14	Elijah	6714	
16	15	Grace	5432	
17	16	Benjamin	7865	
18	17	Chloe	9182	
19	18	Samuel	5623	
20	19	Lily	8749	
21	20	Mason	6021	
22				

Now you need to **import** the data into the Table. Do this by double-clicking the Data Table and pressing the **Import button** and selecting your file. It should then bring in the data in the excel to Unreal Engine.



Now we have the Data table set up it is time to make the binary tree. **Head back to your BinaryTreeNode C++ Class.**

Creation of The Binary Tree Node Class

In the class, you now need to store the data on each node that is going to be in the tree. We are going to store the **Left and Right Binary Tree Node** as well as the **Scoreboard Data** on each. Finally, we need to make a constructor that takes these values as inputs.

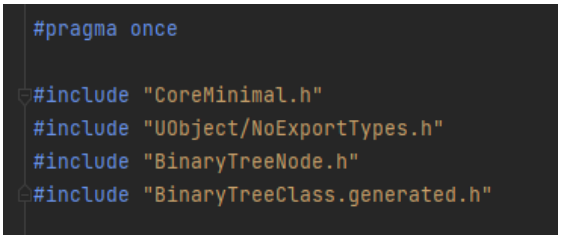
```
UCLASS()
0 derived blueprint classes
class ADVANCECPLUSPLUS_API UBinaryTreeNode : public UObject
{
    GENERATED_BODY()
public:
    UBinaryTreeNode();
    UPROPERTY()
    UBinaryTreeNode* LeftChild;
    UPROPERTY()
    UBinaryTreeNode* RightChild;
    UPROPERTY()
    FScoreboardData Value;

    UBinaryTreeNode(FScoreboardData NewValue) : LeftChild(nullptr), RightChild(nullptr), Value( NewValue) {}

};
```

Creation of Binary Tree Class

Now make **a New C++ Class** that Inherits from **UObject** called **BinaryTreeClass**. Make sure to include your **BinaryTreeNode.h**



```
#pragma once

#include "CoreMinimal.h"
#include "UObject/NoExportTypes.h"
#include "BinaryTreeNode.h"
#include "BinaryTreeClass.generated.h"
```

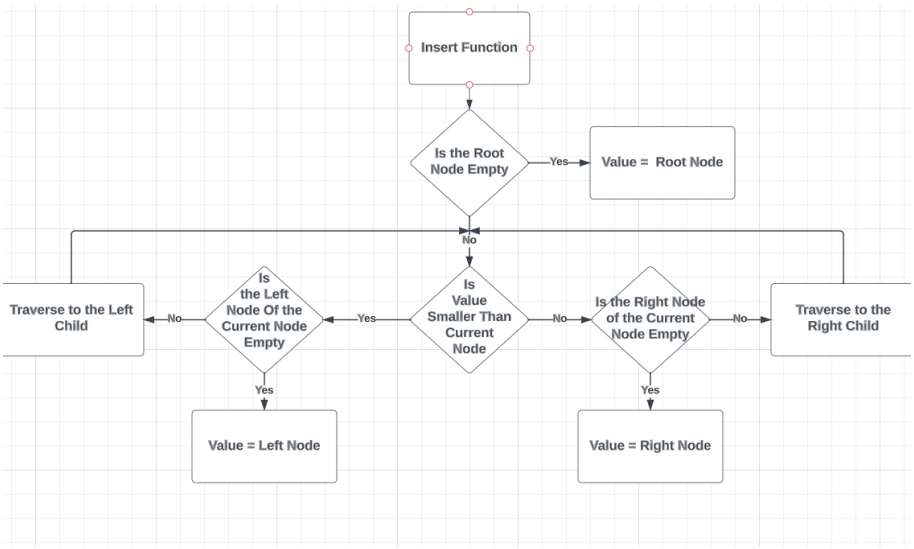
The First **variable** that we are going to need is a **Root**. This will be of type **UBinaryTreeNode***. This will store the **entry point** of are Binary Tree.

We need to **create** a **constructor** that will set this to null when created as it will not always be set when the Binary Tree is Created.

The first Function that we will need to Create is the **Insert Function**. The Insert Function will need an input variable of **FScoreboard Data**.

```
10  UCLASS()
    0 derived blueprint classes
11  class ADVANCECPLUSPLUS_API UBinaryTreeClass : public UObject
12  {
13      GENERATED_BODY()
14  public:
15      UPROPERTY()
16      UBinaryTreeNode* Root;
17
18      UBinaryTreeClass() : Root(nullptr) {}
19
20      void Insert(FScoreboardData Value);
21
```

In the **CPP** add the **insert function** we will need to check where **to place the new data** inputted because **binary trees store the data in size order**, this is done by running a **recursive funct**ion until the place in the tree is found. This diagram shows the checks the code is doing.



```cpp
void UBinaryTreeClass::Insert(FScoreboardData Value)
{
    if(Root == nullptr) //Checks on the Root Being Null
    {
        Root = NewObject<UBinaryTreeNode>( Outer: this);
        Root->Value = Value;
    }
    else //Node isnt Null
    {
        UBinaryTreeNode* CurrentNode = Root;
        while (CurrentNode) //Recursive Loop
        {
            if(Value.Score < CurrentNode->Value.Score) //Checks is value is Smaller
            {
                if(CurrentNode->LeftChild == nullptr) //Checks if there a left node
                {
                    CurrentNode->LeftChild = NewObject<UBinaryTreeNode>( Outer: this); //If empty its assigns the left node to it
                    CurrentNode->LeftChild->Value = Value;
                    return;
                }
                else //If there is a node it then traverses over to it
                {
                    CurrentNode = CurrentNode->LeftChild;
                }
            }
            else //Else it is a Bigger
            {
                if (CurrentNode->RightChild == nullptr) //Checks for right Child
                {
                    CurrentNode->RightChild = NewObject<UBinaryTreeNode>( Outer: this); //If Empty Assigns the right node
                    CurrentNode->RightChild->Value = Value;
                    return;
                }
                else //If there is a node it then traverses over to it
                {
                    CurrentNode = CurrentNode->RightChild;
                }
            }
        }
    }
}
```

It is a lot of code to understand, so make sure to read through the comments to see which parts of the code are linked to the flow diagram. Think of it as a filter that finds the right place for the data.

The following functions that need to be added to the **BinaryTreeClass** are the **InOrderedTraversial** and **ReverseOrderTranversal**. This function take the Root Node and a TArrayPointer of FScoreboardData that it will store all the data in either in order or in reverse order.

```cpp
void InOrderTraversal(UBinaryTreeNode* Node, TArray<FScoreboardData>& Result);

void ReverseOrderTraversal(UBinaryTreeNode* Node, TArray<FScoreboardData>& Result);
```

The functions in C++ are opposites of each other. In OrderTraversal will first look down the **Left Child Nodes and then to the right**. And the Reverse Order, one will look from **Right to Left**. This will then add them to the Array that was inputted.

```cpp
/**
 * @brief Binary Search Because the tree is ordered by the Ints in Struct "FScoreboardData"
 * @param Node (Input Root of the Binary Tree)
 * @param SearchInt (Int Searching For)
 * @return (Returns the Node that is linked to it. Has Struct so links the Name and Score)
 */
UBinaryTreeNode* UBinaryTreeClass::BinarySearch(UBinaryTreeNode* Node, int SearchInt)
{
    if (Node == nullptr || Node->Value.Score == SearchInt)
    {
        return Node;
    }

    if(SearchInt < Node->Value.Score)
    {
        return BinarySearch(Node->LeftChild , SearchInt);
    }
    else
    {
        return BinarySearch(Node->RightChild, SearchInt);
    }
}
```

The final Functions that need to be implemented are InOrderSearch and The BinarySearch; this will take input from the Root Node for the binary tree and the search input. For the Ordered Search, this is a string; for the Binary Search, it is an int. Its return type will be a UBinaryTreeNode*.

The **binary tree search is more efficient** as it can isolate the node the int is on quicker and with fewer searches. **It does this by splitting the tree seeing if the value is higher or low.**

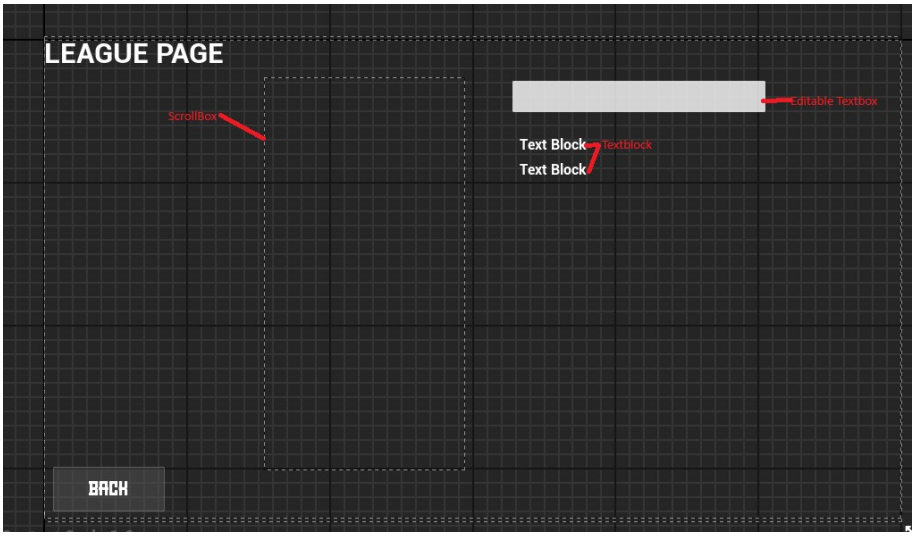
```cpp
/**
 * @brief This is for when searching by Name <- Goes through the whole tree Left to Right As names arent in order
 * @param Node (Input Root of the Binary Tree)
 * @param SearchString
 * @return (Returns the Node that is linked to it. Has Struct "FScoreboardData" so links the Name and Score)
 */
UBinaryTreeNode* UBinaryTreeClass::OrderedSearch(UBinaryTreeNode* Node, FString SearchString)
{
    if(Node == nullptr)
    {
        return nullptr;
    }
    if(Node->Value.Name == SearchString)
    {
        return Node;
    }
    UBinaryTreeNode* leftResult = OrderedSearch(Node->LeftChild, SearchString);
    UBinaryTreeNode* rightResult = OrderedSearch(Node->RightChild, SearchString);
    if (leftResult != nullptr)
    {
        return leftResult;
    }
    else
    {
        return rightResult;
    }
}
```

After all these functions have been implemented, you have completed the set up to the Binary Tree Class.

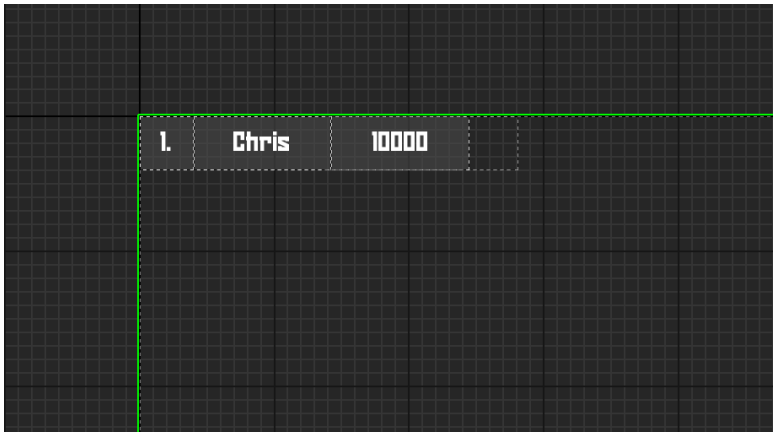
Menu Creation

It is time to make the menu that is going to display the data to do this you need to make 2 C++ Classes the Inherit from UMG widget like we did in the last tutorial. One is the **LeagueMenu** and one that is the **LeagueTemplate**.

Make both a **Blueprint Derived Class and open the UMG editor**. In the League Menu screen, make sure to have **2 Textblocks** where we will output the data, an **editable textbox** for the input of the data, and a Scrollbox where we will add menu data.



In the **LeagueTemplate**, we will make **one version of the table** that will be **repeated**. Make sure to **include 3 TextBlocks** where we can output are information. I added a background to make the table look better. You can do the same.



Scoreboard Manager Creation

The next step is **controlling the Menu** based of the data. This will be done through **a controller system**. Create a C++ class based of an actor. Name it **ScoreboardManager.** I will run through the variables.

Variables

Type of Data	Name	Reason
UDataTable*	ScoreboardDataTable	Holds the data table to read and implement into a binary tree
UScrollBox*	Scrollbox	Holds a reference to the scroll box where we will add templates.
TSubClassOf<UUserWidget>	Template Widget	Holds a reference to the Template that we made to spawn more in and manipulate the text inside.
FScoreboardData	Output Text	Variable to hold the output from reading the binary tree.
UBinaryTreeClass*	ScoreboardBinaryTree	Holds the Binary Tree

```
/**
 * Please Assign this in the Inspector of BP_ScoreboardManager <- Inserted Data CSV File To Data Table
 */
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "ScoreboardDataTable")
UDataTable* ScoreBoardDataTable;

//Variable to Create the Scroll Box and the Template, Function to Insert them Called once Scroll Box is Assigned <- Blueprint Called in ScoreBoard Manager
UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Widgets")
UScrollBox* ScrollBox;

UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Widgets")
TSubclassOf<UUserWidget> TemplateWidget;

UPROPERTY(BlueprintReadOnly)
FScoreboardData OutputText;

UPROPERTY()
UBinaryTreeClass* ScoreBoardBinaryTree;
```

Functions

Name	Input	Return Type	Notes
CreateTemplate	None	Void	Creates and edits the template to the scroll box
SearchBinaryTree	Int	Void	Used as an overload function to run the binary tree search
SearchBinaryTree	String	Void	This is an overload function to run the in-order search.
ResultHander	UBinaryTreeNode*	Void	Takes the output of the search and handles the changing on the output text.
BeginPlay	None	Virtual Void	This is inputted by default but will be used to transfer data to the binary tree
InitSearch	FString	Void	Blueprint callable function that get the inserted word in the text box

```
37
38          UFUNCTION(BlueprintCallable)
39          void CreateTemplate();
40
41          //Overloaded Functions to Split the Int and the String Searches
42          void SearchBinaryTree(int SearchItem);
43          void SearchBinaryTree(FString SearchItem);
44
45          UFUNCTION()
46          void ResultHandler(UBinaryTreeNode* Result);
47
48          UFUNCTION(BlueprintCallable)
49          void InitSearch(FString SearchItem);
50
51      protected:
52          virtual void BeginPlay() override;
53
```

BeginPlay

The first function we are going to look at is the BeingPlay()

Step 1: Create the BinaryTreeClass and assign it to the ScoreBoardBinaryTree

Step 2: Check the Data Table isn't empty.

Step 3: Make a TArray that is filled with the Datatables names.

Step 4: For loop that loops through the whole data table.

Step 5: Get the data from each row via the Name of the Row

Step 6: Create a temp FScoreboardData and assign the values.

Step 7: Insert this FscoreboardData to the binary Tree.

```
void AScoreboardManager::BeginPlay()
{
    Super::BeginPlay();
    ScoreBoardBinaryTree = NewObject<UBinaryTreeClass>(Outer: this); //Creates Binary Tree For this to Use

    if (ScoreBoardDataTable) //Imports Data from Datatable into the Binary Tree (InOrder)
    {
        TArray<FName> RowNames = ScoreBoardDataTable->GetRowNames();
        for (FName RowName : RowNames)
        {
            FScoreboardData* DataRow = ScoreBoardDataTable->FindRow<FScoreboardData>(RowName, ContextString: "");
            if (DataRow)
            {
                FScoreboardData ScoreboardTemp;
                ScoreboardTemp.Name = DataRow->Name;
                ScoreboardTemp.Score = DataRow->Score;
                ScoreBoardBinaryTree->Insert( ScoreboardTemp);
            }
        }
    }
}
```

CreateTemplate

Check if the **scroll box has been assigned**. Then, create a **TArray** for the data to be transferred to **InOrder**. Next run the **Reverse Order Function** that we created and pass in the **root of the tree** and **the array.**

You then want to run a loop the **length of the returned array**. In the loop turn the data into a **FScoreboard Data from** the array**. Next, create a new widget declare in the ULeagueTemplate widget as the class and pass it** in the **GetWorld** function and the **Template Widget.** Next run the **ChangeText** function and pass in the data. Finally add it **the Scroll Box as a child.**

```
void AScoreboardManager::CreateTemplate()
{
    if(ScrollBox)
    {
        TArray<FScoreboardData> InOrderResult; // <- Returns Array of them in order OR in this case reversed HI-Score
        ScoreBoardBinaryTree->ReverseOrderTraversal(ScoreBoardBinaryTree->Root, [&] InOrderResult);

        for (int32 i = 0; i < InOrderResult.Num(); ++i) //Creates the Template and Changes the Text in The Widget. Does this for each entry
        {
            FScoreboardData Value = InOrderResult[i];
            ULeagueTemplate* NewItem = CreateWidget<ULeagueTemplate>( OwningObject: GetWorld(), TemplateWidget);
            NewItem->ChangeText(Value.Name, Value.Score, Place: i);
            ScrollBox->AddChild(NewItem);
        }
    }
}
```

InitSearch

In the **InitSearch** Function we are **filtering** to see if the **input is numeric**. To do this get the inputed Fstring and run a IsNumeric check. If it is **covert it to a int** and pass it into the **SearchBinaryTree** function. If it isn't don't change the string and pass it to the **SearchBinaryTree** function as it is an **overloaded function.**

```
void AScoreboardManager::InitSearch(FString SearchItem)
{
    if(SearchItem.IsNumeric())
    {
        int temp = FCString::Atoi(*SearchItem);
        SearchBinaryTree(temp);
    }
    else
    {
        SearchBinaryTree(SearchItem);
    }
}
```

SearchBinaryTree (Overloaded Function)

In the function call the corresponding function. For an int call the binary search and for text call the ordered search function. Finally pass the results into the result handler.

```
void AScoreboardManager::SearchBinaryTree(int SearchItem)
{
    UBinaryTreeNode* Result = ScoreBoardBinaryTree->BinarySearch(ScoreBoardBinaryTree->Root , SearchItem);
    ResultHandler(Result);
}

void AScoreboardManager::SearchBinaryTree(FString SearchItem)
{
    UBinaryTreeNode* Result = ScoreBoardBinaryTree->OrderedSearch(ScoreBoardBinaryTree->Root , SearchItem);
    ResultHandler(Result);

}
```

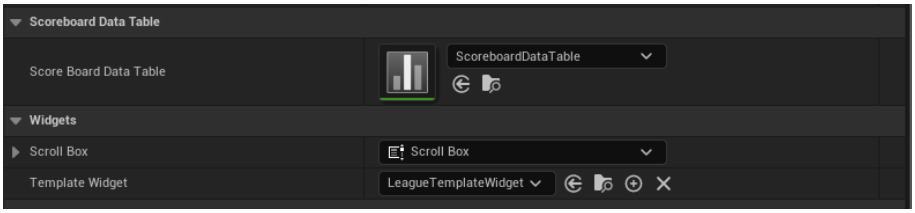
Results Handler

In the results handler check if a result has been passed in and set the output text to the value that is passed in, make sure to have a else statement that resets when there isn't a player with the same inputed.

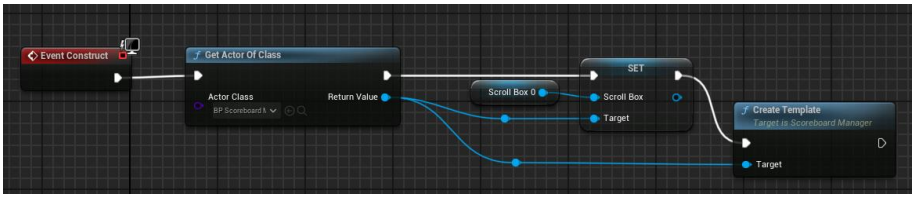
```
void AScoreboardManager::ResultHandler(UBinaryTreeNode* Result)
{
    if(Result)
    {
        OutputText.Name = Result->Value.Name;
        OutputText.Score = Result->Value.Score;
    }
    else
    {
        OutputText.Name = "There is No Player With This Name";
        OutputText.Score = 0;
    }
}
```

Blueprint

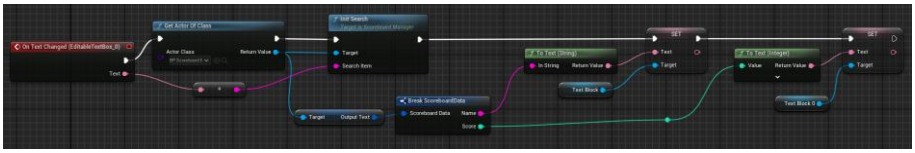
Make a blueprint variant of the C++ Class and name it BP_ScoreManager. Make sure to assign the variable in the inspector.



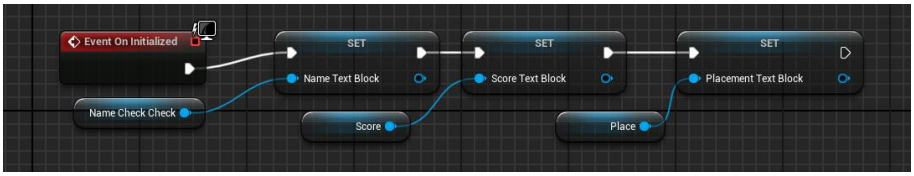
In the blueprint of LeagueMenu on the construct function get the scoreboard class and set the scroll box variables. This needs to be here due to the scroll box being destroyed when unloading the menu. Then run the Create Template function.



In this blueprint as well you need to make a OnTextChange function from the editable text box. You need to find the Scoreboard Manager and then run the Init search function as well as link the Output Text variable and assign it to the text box in the menu.



In the League Template you need to pass in the variables for the Name, Score and Placement you need to make sure this text is set as a variable.



Also you need to set up a function to remove it from the scene when destruct is called as it is a child of the scoreboard

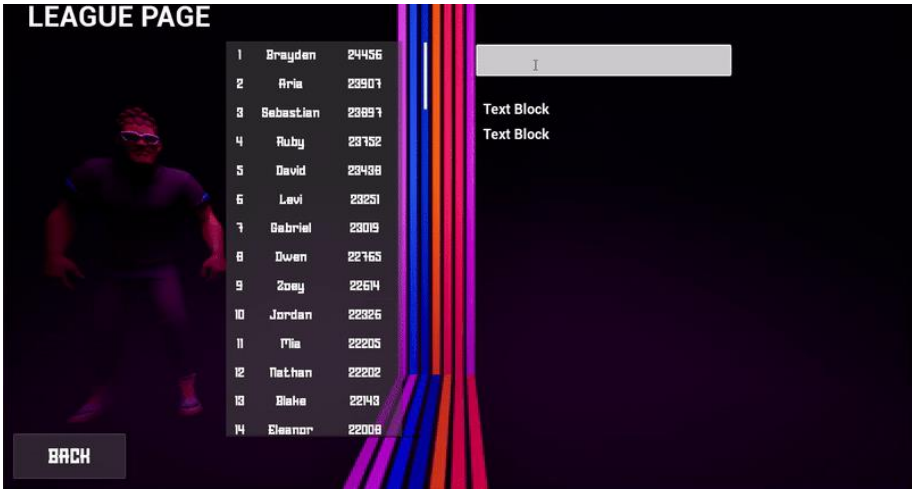
These Function in C++ will look like this

```cpp
#include "LeagueTemplate.h"

void ULeagueTemplate::ChangeText(FString Name, int Score, int Place)
{
    if (NameTextBlock)
    {
        NameTextBlock->SetText(FText::FromString(Name));
        ScoreTextBlock->SetText(FText::FromString(FString::Printf(Fmt: TEXT("%d"), Score)));
        PlacementTextBlock->SetText(FText::FromString(FString::Printf(Fmt: TEXT("%d"), Place + 1)));

    }
}
void ULeagueTemplate::RemoveTemplate()
{
    RemoveFromParent();
    ConditionalBeginDestroy();
}
```

Once this is all done it should be all linked and should now be able to run the code. Make sure to compile and save all blueprint and scripts. **Congratulations.**



Extension Task

Try adding more data to the binary tree at runtime to create a scoreboard that updates. All the functionality is there. A hint will be to make another input box and once filled in to add it to the tree. You many want to do this on a button press. You will also need to refresh the template boxes. You make made the functionality for this already.

Appendix 3: Link List Creation (Tutorial 3)

Linked List / UE5.3.2

Contents

Introduction

This tutorial will continue looking at advanced C++ programming and creating them in Unreal Engine. In this tutorial we are going to make a Linked List that will display a equip system in a battle pass so the user can change the currently equipped weapon in game. Please complete the previous 2 tutorials not because we are using the code but the fundamentals of menu creation will be used.

Requirements

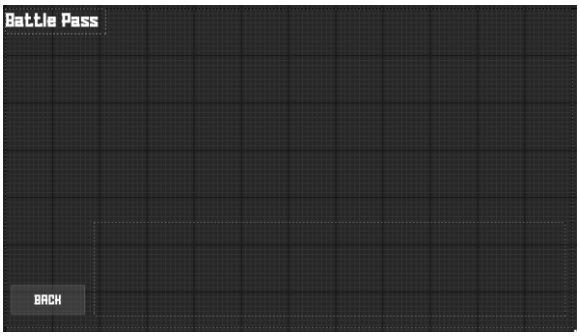
This should work on any version of Unreal Engine but for this project it has been tested on **5.3.2**.

Make sure to have **Rider**, Visual Studio, Or an IDE of your choosing installed. In this case **Rider** is used.

Menu System

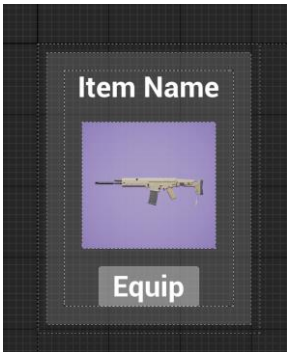
We are going to follow the same principles of making a scroll box and inserting a template into it and changing the data inside. First of all you want to make a new UI Widget as a C++ Class and name BattlePassMenu, and another called BattlePassTemplate. This is the example that we are going to be using. Make them into a Blueprint version after and design them to include a scroll box.

BattlePassMenu Design



BattlePassTemplate Design

Make sure to include a Item Name, Image and a button to equip the weapon



Linked List Class

As we have done before we are going to make a base class of a UObject. This should be called LinkedListClass. In this script you need to make a struct called FNode. This node will be storing all the data as well as the pointers to the next item in the list. This should be placed outside the UObject class.

```
struct FNode
{
public:
    FBattlePassDataStruct Data;
    FNode* Next;
    FNode(FBattlePassDataStruct InData) : Data( InData), Next(nullptr) {}
};
```

We will also need to make a struct that will hold all the data on the items. This should be called FBattlePassDataStruct. This should include a UTexture2D that will store the image, a FString that holds the name, a Boolean property that holds the unlock status and a UMaterial that holds the gun

texture. Make sure to include the USTRUCT at the top or blueprint will not be able to access this data.

```
USTRUCT(BlueprintType)
struct FBattlePassDataStruct
{
    GENERATED_BODY()
public:
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UTexture2D* Image;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    FString Name;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    bool Unlocked;
    UPROPERTY(EditAnywhere, BlueprintReadWrite)
    UMaterial* GunTexture;
};
```

In the body of the Linked List Class you need to make a variable of the FNode and name it First as this is are access point to the list. There also needs to be a constructor that sets the First to nullptr as you make create a linked list before assigning the first value.

```
UCLASS()
0 derived blueprint classes
class ADVANCECPLUSPLUS_API ULinkedListClass : public UObject
{
    GENERATED_BODY()
public:
    FNode* First;
    ULinkedListClass() : First(nullptr) {}
    void AddNode(FBattlePassDataStruct Data);
    void InsertAtBeginning(FBattlePassDataStruct Data);
    void DeleteAtPosition(int Pos);
    int GetPosition(FString Name);
};
```

Function Description

Functions Name	Return Type	Input Class	Description
AddNode	Void	FBattlePassDataStruct	Inserts node at end of the list
InsertDataAtBeginning	Void	FBattlePassDataStruct	Inserts node at the start of list
DeleteAtPosition	Void	Int	Deletes Data at int position in the list
Int GetPos	Int	Fstring	Gets the passion of the data using the FString Name

Function Implementation

AddNode()

The add node function first creates a new node using the data provided. It then checks the first place on the linked list and if it empty will fill the first place. Else it will go through the Linked List by making the current node the one it is checking and seeing if it has a “Next” place if it doesn’t it will the assign the Next node to the inputted data.

```
void ULinkedListClass::AddNode(FBattlePassDataStruct Data)
{
    FNode* NewNode = new FNode(Data);
    if (!First)
    {
        First = NewNode;
    }
    else
    {
        FNode* CurrentNode = First;
        while (CurrentNode->Next)
        {
            CurrentNode = CurrentNode->Next;
        }
        CurrentNode->Next = NewNode;
    }
}
```

InsertDataAtBeginning()

Inserting the node at the beginning is very simple. It first makes the data inputted into a new node. It then assigns the new nodes next pointer to the first of the current linked list and then sets the new node to the first in the chain.

```
void ULinkedListClass::InsertAtBeginning(FBattlePassDataStruct Data)
{
    FNode* newNode = new FNode(Data);
    newNode->Next = First;
    First = newNode;
}
```

DeleteAtPosition()

This function is more complicated. It first checks there is a valid first entry into the linked list. If the inserted int is zero it will then set the first point to a temporary variable and traverse to the next on of the list and set that as the “First” node and after delete the temporary node.

```
void ULinkedListClass::DeleteAtPosition(int Pos)
{
    if (Pos < 0)
    {
        UE_LOG(LogTemp, Warning, TEXT("Invalid position"));
        return;
    }

    if (Pos == 0)
        {
            if (First != nullptr)
            {
                FNode* temp = First;
                First = First->Next;
                delete temp;
            }
            else
            {
                UE_LOG(LogTemp, Warning, TEXT("List is empty"));
            }
    }
```

If the data being deleted isn't the first node it will set the first node to the current position and then will use a while loop to traverse down the linked list it will save the previous node. Once it reaches the position it was meant to it will then run the if statement to check it not null and will the set the previous node to the current node and then delete the current node. There is then a check if it the number input is too large for the list and will output a log.

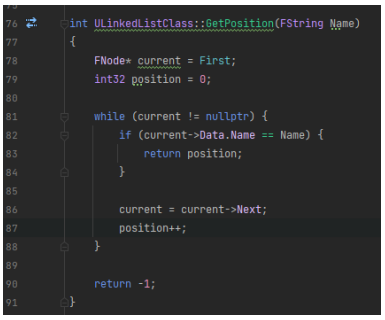
```
else
{
    FNode* current = First;
    FNode* previous = nullptr;
    int32 currentPosition = 0;

    while (current != nullptr && currentPosition < Pos)
    {
        previous = current;
        current = current->Next;
        currentPosition++;
    }

    if (current != nullptr)
    {
        previous->Next = current->Next;
        delete current;
    }
    else
    {
        UE_LOG(LogTemp, Warning, TEXT("Position exceeds the length of the list"));
    }
}
```

GetPosition()

Get position uses a while loop to traverse the linked list and compares the inputed data to the Fstring that is part of the struct. It does this a counts the loop. Once it finds this it returns the loop if not it returns a negative integer. That is the Link List Class setup.



```
int ULinkedListClass::GetPosition(FString Name)
{
    FNode* current = First;
    int32 position = 0;

    while (current != nullptr) {
        if (current->Data.Name == Name) {
            return position;
        }

        current = current->Next;
        position++;
    }

    return -1;
}
```

BattlePassManager Class

We are going to use the same system of a controller that we have before to access the link list class. To do this make a C++ that inherits from the AActor Class and make a Blueprint derived class from this. In the C++ we are going to need several functions and variables listed below.

Functions	Returning	Inputs	Description
CreateTemplate	Void	Nothing	This will create the templates for the data to be passed in
HoverTextureChange	Void	UMaterial*	Function that will change when hovering over a button
ButtonTextureAssign	Void	UMaterial*	Function that will assign the texture to the gun

UnHoverTextureChange	Void	Nothing	Function that will change the texture back to default.
LoopWithDelay();	void	Nothing	Function that delays the spawn time to create an animation effect of the UI

Variable	Type	Description
BattlePassLinkedList	ULinkedListClass*	Holds the Linked List with all the data
BattlePassArray	TArray<FBattlePassDataStruct>	Holds the initial data for which the link list reads
ScrollBox	UScrollBox*	Holds the reference to the scrollbox
TemplateWidget	TSubclassOf<UUserWidget>	Reference to the template class to spawn in
EquippedGunTexture;	UMaterial*	Local Gun texture to refer to when applying the texture
CurrentIndex	Int	Used for creating the templates

```
public:
    ABattlePassManager();

    UPROPERTY()
    ULinkedListClass* BattlePassLinkedList;

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "BattlePass")
    TArray<FBattlePassDataStruct> BattlePassArray;  Unchanged in assets

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Widgets")
    UScrollBox* ScrollBox;  Unchanged in assets

    UPROPERTY(EditAnywhere, BlueprintReadWrite, Category = "Widgets")
    TSubclassOf<UUserWidget> TemplateWidget;  Unchanged in assets

    UFUNCTION(BlueprintCallable)
    void CreateTemplate();

    UFUNCTION(BlueprintCallable)
    void HoverTextureChange(UMaterial* Texture);

    UFUNCTION(BlueprintCallable)
    void ButtonTextureAssign(UMaterial* Texture);

    UFUNCTION(BlueprintCallable)
    void UnHoverTextureChange();

    UPROPERTY()
    UMaterial* EquippedGunTexture;

    void LoopWithDelay();

    int CurrentIndex;
```

BeginPlay()

The begin play function needs to create the linked list and assign the equipped gun a value.

```
void ABattlePassManager::BeginPlay()
{
    Super::BeginPlay();
    BattlePassLinkedList = NewObject<ULinkedListClass>( Outer: this);
    EquippedGunTexture = BattlePassArray[0].GunTexture;
}
```

CreateTemplate()

When the create template is called it will check the scroll box is set and set the index to 0 and then call the function LoopWithDelay.

```
void ABattlePassManager::CreateTemplate()
{
    if(ScrollBox)
    {
        CurrentIndex = 0;
        LoopWithDelay();
    }
}
```

LoopWithDelay()

In this function it will get the length of the battlepass array and loop that many times. It will get the data from the array and create a local value. It then makes a template like we did the previous tutorial. It then calls the function on the Widget to change the values and adds it as a child to the scroll box. It then waits for 0.1 seconds to run the function again using the time manager.

```
void ABattlePassManager::LoopWithDelay() //Adds Animation to the Boxes Appearing
{
    if (CurrentIndex < BattlePassArray.Num())
    {
        FBattlePassDataStruct Value = BattlePassArray[CurrentIndex];
        UBattlePassTemplate* NewItem = CreateWidget<UBattlePassTemplate>(OwningObject GetWorld(), TemplateWidget);
        NewItem->ChangeItems(Value.Name, Value.Image, Value.Unlocked, Value.GunTexture);
        ScrollBox->AddChild(NewItem);
        CurrentIndex++;
        if (CurrentIndex < BattlePassArray.Num())
        {
            GetWorldTimerManager().SetTimer( [&] TimerHandle, InObj this, &ABattlePassManager::LoopWithDelay, InRate 0.1f, InbLoop false);
        }
    }
}
```

Hover() and Unhover()

In these functions they set the model to the current texture or back to the default.

```
void ABattlePassManager::HoverTextureChange(UMaterial* Texture)
{
    AGunRender* GunRenderActor = Cast<AGunRender>( Src UGameplayStatics::GetActorOfClass(GetWorld(), AGunRender::StaticClass()));
    GunRenderActor->ChangeGunTexture(Texture);
}
```

```
void ABattlePassManager::UnHoverTextureChange()
{
    AGunRender* GunRenderActor = Cast<AGunRender>( Src UGameplayStatics::GetActorOfClass(GetWorld(), AGunRender::StaticClass()));
    GunRenderActor->ChangeGunTexture(EquippedGunTexture);
}
```

ButtonTextureAssign()

This changes the local to the texture passed in.

```
33      void ABattlePassManager::ButtonTextureAssign(UMaterial* Texture)
34      {
35          EquippedGunTexture = Texture;
36      }
```

BattlePassTemplate Class

One of the function is ChangeItem sets the variables passed in into the local variables that hold them. A UTextblock, UImage and UMaterial that are used in blueprint.

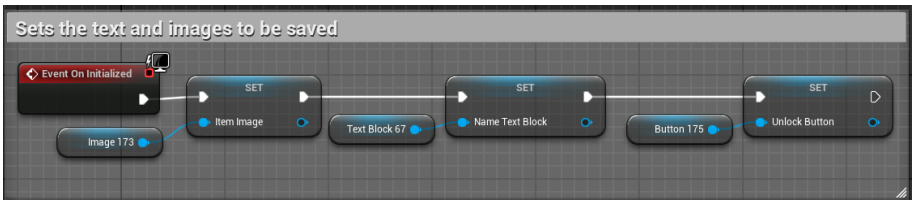
```
void UBattlePassTemplate::ChangeItems(FString Name, UTexture2D* Item, bool Unlock, UMaterial* Texture)
{
    NameTextBlock->SetText(FText::FromString(Name));
    ItemImage->SetBrushFromTexture(Item);
    TextureMaterial1 = Texture;
}
```

The second is a remove template function for when it is being destroyed.

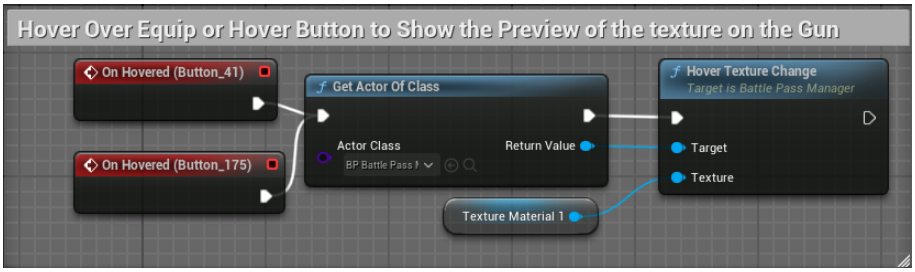
```
void UBattlePassTemplate::RemoveTemplate()
{
    RemoveFromParent();
    ConditionalBeginDestroy();
}
```

BattlepassTemplate Blueprint

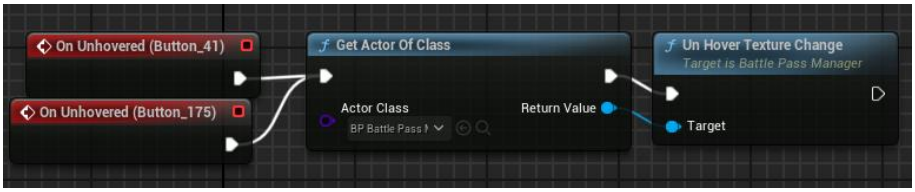
On the Initialize it needs to set the C++ variable to the blueprint variants from the UI graph.



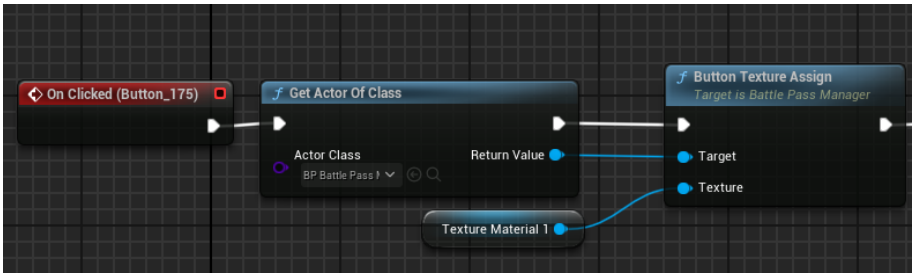
When the button is hovered over to equip or view the skin it should run the on hover function and pass in the local material.



It should then run the unhover function when the cursor has left the button.

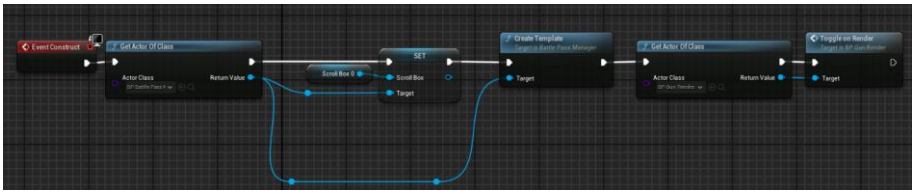


On the button click it should then run the Button texture assign and pass in the material

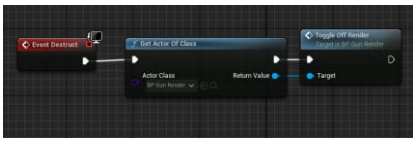


Battle Pass Manager Blueprint

On the construction function it should get the battle pass manager and assign the scrollbox to the variable in the script, it should then run the create template function. Finally it should find the gun render and toggle it render.

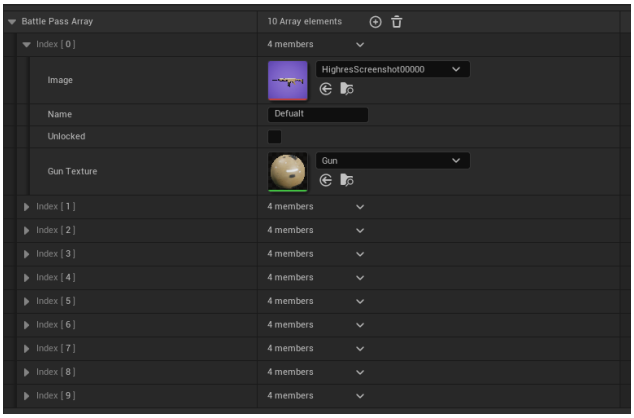


On the destruct it should find the gun render and disable the render

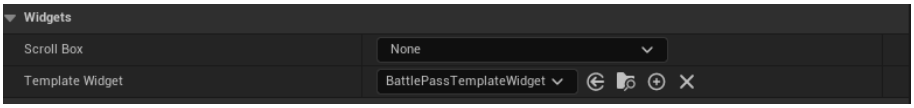


In Engine Setup

In the engine you need to add all the content to the Battle Pass Array. This includes a screenshot of the gun the name the unlock status and the gun texture.



You also need to add the template to the battle pass manager, or it will not spawn in.



Gun Render Class

After this insert a new C++ class that is called AGunRender that is inherited from AActor. In this header file add a reference to the Static Mesh and a function called GunChangeTexture that has a UMaterial Passed in

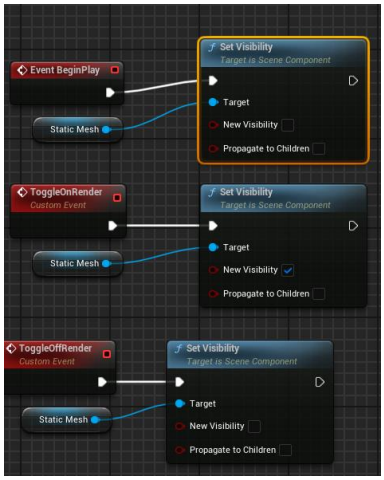
```
public:
    UPROPERTY(EditAnywhere)
    UStaticMeshComponent* GunMesh;  Unchanged in assets

    UFUNCTION(BlueprintCallable)
    void ChangeGunTexture(UMaterial* InputTexture);
```

In the function for Change Texture you need to find the component of the Gun Mesh then check that it is valid and the input texture is valid then set the material.

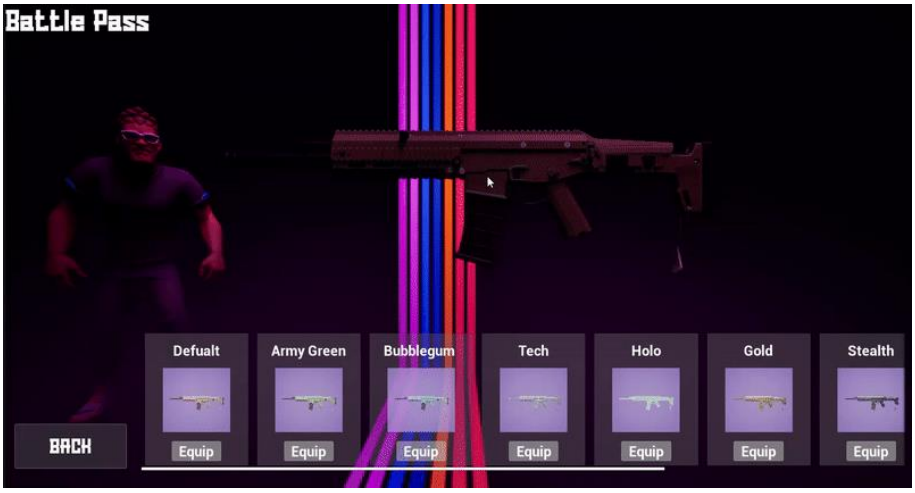
```
void AGunRender::ChangeGunTexture(UMaterial* InputTexture)
{
    GunMesh = FindComponentByClass<UStaticMeshComponent>();
    if (GunMesh && InputTexture)
    {
        GunMesh->SetMaterial( ElementIndex: 0, InputTexture);
    }
}
```

Finally in the blueprint you need to set the visibility of the gun renderer when it starts and when the Renderer is toggled. Make sure to pass in the static mesh.



Make sure to Compile and save all your work.

Now that you have completed all the steps and linked all the blueprint and C++ you should now have a project that looks similar to this.



Extension Task

Use the data saving system in unreal engine to load this gun into a playable map. This will use the USaveGameData function similar to unitys player prefs functions.

[Saving and Loading Your Game | Unreal Engine 4.27 Documentation]

This is a link to the documentation that covers all the code that you will need. Here is an example video.