Game Programming using C++

SDC GPWC,
Faculty of Engineering and Technology (ITER)
SOA (Deemed to be) University, BBSR, Odisha
2025

Name of the Subject	**Game Programming with C++**
Subject Code	CSE 3545
Grading Pattern	1
Subject Credits	4
Course	CSE, 6th Semester (2022 Batch)

Introduction to course

To equip students with the foundational and advanced concepts of game development using C++, including the application of object-oriented programming, game physics, graphics rendering, artificial intelligence, and gameplay mechanics. The course aims to develop students' problem-solving and creative skills by designing and implementing interactive and immersive gaming experiences. Students will also learn to use industry-standard tools and libraries to create efficient and scalable game applications.

Program Outcomes (POs)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis:** Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions:** Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems:** Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage:** Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modelling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society:** Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability:** Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics:** Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and teamwork:** Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication:** Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

11. **Project management and finance:** Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning:** Recognize the need for and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Specific Outcomes (PSOs)

1. **PSO 1:** The ability to understand, analyze and develop computer programs in the areas related to business intelligence, web design and networking for efficient design of computer-based systems of varying complexities.
2. **PSO 2:** The ability to apply standard practices and strategies in software development using open-ended programming environments to deliver a quality product for business success.

Course Outcome (COs)

Students will be able to

1. **CO1: write**, **compile** and **execute** simple C++ programs to solve computational problems.
2. **CO2: become familiar** with basic games using SFML and Visual Studio, implementing sprites, animations, player inputs, and Heads-Up Display (HUD) elements.
3. **CO3: implementing** Game Mechanics and Physics like collision detection, pickups, bullets, and sound effects while applying dynamic collision detection and basic physics concepts.
4. **CO4: develop** advanced game features, such as zombie shooter games, using sprite sheets, vertex arrays, texture management, and layering views.
5. **CO5: apply** advanced object-oriented programming concepts like inheritance, polymorphism, and abstraction to improve code management and game scalability.
6. **CO6: implement** file I/O, sound effects, and refining the game's HUD and UI, while optimizing performance using C++ references, pointers, and the Standard Template Library (STL).

Course Structure

Grading Pattern

Type	Category		Mark wait	
	Attendance (if >75):		**05**	
INTERNAL	Assignments and Quizzes :		**20**	
	Mid-term:		**15** ------->	Mapped from 30
- - - - - - -	- - - - - - - - - - - - - - - -	- - - - - -	- - - - - - -	- - - - - - - - -
	End-term:	LAB test	**15**	
EXTERNAL		END-sem	**45** ------->	Mapped from 60
		Total	**100**	

Credits and Course Format

- Grading Pattern: 1
- Credits: 4
- 3 Classes/week, 1hr/class
- 1 Labs/week, 2hr/Lab

Course Curriculum

CSE 3545	Game Programming with C++	4	1
C++, SFML, Visual Studio, and Starting the First Game, Variables, Operators, and Decisions – Animating Sprites, C++ Strings and SFML Time – Player Input and HUD, Loops, Array, Switches, Collisions, Sound, and End Conditions – Making the Game Playable, Object-Oriented Programming – Starting the Pong Game, Dynamic Collision Detection and Physics – Finishing the Pong Game, SFML Views – Starting the Zombie Shooter Game, C++ References, Sprite Sheets, and Vertex Arrays, Pointers, the Standard Template Library, and Texture Management, Collision Detection, Pickups, and Bullets, Layering Views and Implementing the HUD, Sound Effects, File I/O, and Finishing the Game, Abstraction and Code Management – Making Better Use of OOP, Advanced OOP – Inheritance and Polymorphism.		Textbook – Beginning C++ Game Programming by Horton, Packt Publishing Weekly Course Format: 3L - 2P	

Books for reference

1. **J. Horton, Beginning C++ game programming: learn to program with C++ by building fun games, Second edition. Place of publication not identified: Packt Publishing, 2019.**
2. **E. Balagurusamy, Object oriented programming with C++, Eighth Edition. New Delhi: McGraw Hill Education (India) Private Limited, 2021.**

Introduction to Evaluation strategy

Mid-semester Question Format :

- 5 questions each carrying 3 bits
- each bit will be of 2 marks
- Total of 30 marks
- Exam will be conducted for 2 Hrs

External Lab Question Format

- 3 Question Programming Based
- Each carrying 5 Marks
- Total of 15 marks.
- Exam will be conducted for 2 Hrs

End-Semester Question Format

- 10 questions each carrying 3 bits
- each bit will be of 2 marks
- Total of 60 marks
- Exam will be conducted for 3 Hrs

Quiz Test (Minimum 4)

- 30 questions each carrying 1 mark
- Total Mark 30
- Exam will be conducted for 30 Mins
- Will be done in online mode. (Present or absent have to attend the quiz test.)
- Average of all quizzes will be considered for final evaluation.

Assignment Submission

- There will be Minimum 10 assignments which students have to submit each carrying 10 marks.
- The students have to complete the assignment and upload the scanned copy of it in shared google form along with the screenshot of the output.
- At the end of the semester before the External Lab exam the students have to submit the complete assignments in physical format.
- Assignments are to be submitted within the given deadline failing to which 4 marks will be deducted.
- Students may be asked to redo in case any unfair means is observed.

Introduction to C++

Introduction to C++

C++ is a general-purpose, high-performance programming language that supports object-oriented, procedural, and generic programming paradigms. It was developed by **Bjarne Stroustrup** in 1979 as an extension of the C programming language, adding features such as classes and object-oriented programming.

Key Features of C++:

1. **Object-Oriented Programming**: Supports concepts like classes, objects, inheritance, polymorphism, encapsulation, and abstraction.
2. **Portability**: Code written in C++ can run on different platforms with minimal or no modifications.
3. **Efficiency and Performance**: Provides direct access to hardware and system-level resources, making it ideal for performance-critical applications.
4. **Standard Template Library (STL)**: A rich library of templates for data structures (e.g., vectors, lists), algorithms (e.g., sorting, searching), and iterators.
5. **Low-Level Manipulation**: Allows direct memory manipulation using pointers, which is helpful in system programming.
6. **Extensibility**: Users can define their own types and overload operators for custom behavior.
7. **Rich Library Support**: Includes built-in libraries for file handling, data manipulation, and more.

What is the difference between C and C++?

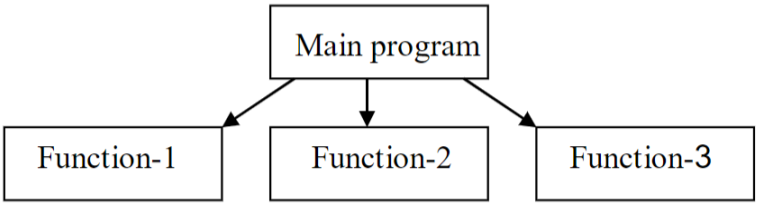
- C is a **procedural programming language**, while C++ is a **multi-paradigm language** that supports both procedural and object-oriented programming.

Why C++ Was Developed?

- C was a popular and efficient programming language for system-level programming, but it lacked features that made it suitable for solving complex, large-scale problems.
- At the same time **Simula** programming language was developed in the 1960s by **Ole-Johan Dahl** and **Kristen Nygaard** at the Norwegian Computing Center in Oslo, Norway. It is widely regarded as the first **object-oriented programming (OOP)** language. Initially designed for simulation applications, Simula allowed users to model real-world systems, such as queuing systems or process flows.
- Bjarne Stroustrup, the creator of C++, acknowledged that Simula's class concept was a key inspiration.

Procedure Oriented Programming vs Object Oriented Programing

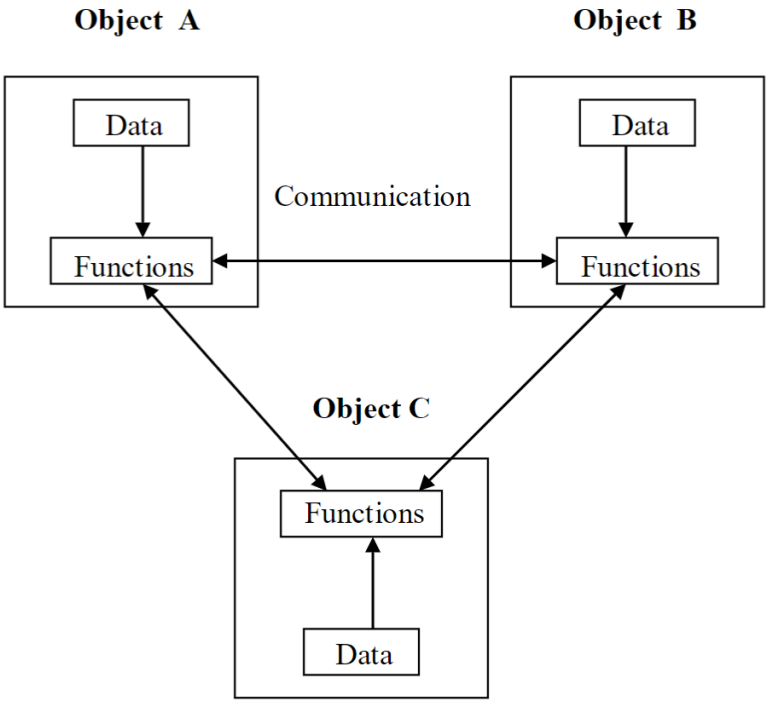
Procedure Oriented Programming (POP) focuses on functions or procedures to achieve tasks. Programs in POP follow a linear and sequential approach, with data being shared globally among functions. This can lead to challenges in maintaining and securing data. Modularity is achieved by dividing the program into smaller, independent functions. However, code reusability is limited, making scalability difficult in larger projects. POP is typically used for simpler applications, with examples of POP languages including C and Fortran.



The disadvantage of the procedure oriented programming languages is:

1. Global data access
2. It does not model real word problem very well
3. No data hiding

On the other hand, **Object Oriented Programing (OOP)** emphasizes objects, which bundle data and behavior together. Key concepts in OOP include classes, objects, inheritance, encapsulation, polymorphism, and abstraction. This approach allows better data security by encapsulating data within objects and restricting access. OOP ensures modularity through classes and objects, improving code reusability and scalability. It closely maps to real-world entities, making it intuitive for designing complex applications. OOP languages like Java, Python, and C++ are ideal for developing enterprise-level software, games, and web applications.



Features of the Object Oriented programming

1. Emphasis is on doing rather than procedure.
2. programs are divided into what are known as objects.
3. Data structures are designed such that they characterize the objects.
4. Functions that operate on the data of an object are tied together in the data structure.
5. Data is hidden and can't be accessed by external functions.
6. Objects may communicate with each other through functions.
7. New data and functions can be easily added.
8. Follows bottom-up approach in program design.

Feature	Procedure-Oriented Programming (POP)	Object-Oriented Programming (OOP)
Focus	Functions/Procedures	Objects
Key Concept	Functions	Classes, Objects
Data Security	Less secure	Encapsulation for security
Modularity	Function-based	Class and Object-based
Reusability	Low	High
Scalability	Difficult	Easier
Real-world Mapping	No	Yes
Example Languages	C, Fortran	Java, Python, C++

Class vs Structure

A **structure** is a user-defined data type that groups related variables (fields) together under one name. It is a value type, typically used for creating lightweight objects that do not require inheritance or complex behavior. Structures are designed to store and manipulate simple data efficiently. They are often used in procedural and object-oriented programming for small, self-contained data entities.

A **class** is a blueprint for creating objects in object-oriented programming. It defines a reference type that encapsulates data (fields) and behavior (methods) into a single entity. Classes are designed for creating complex, reusable, and scalable objects that can interact with each other. They support advanced object-oriented features like inheritance, polymorphism, and encapsulation.

Feature	Class	Structure
Type	Reference	Value
Memory Allocation	Heap	Stack
Inheritance	Supported	Not supported
Default Access	Private	Public
Performance	Slower (heap management)	Faster (stack-based)
Use Case	Complex objects, large data	Lightweight objects, small data
Constructor	Parameterless by default	Explicit, all fields initialized

Some Basic Concepts of OOP

1. **Classes**
   - A **class** is a blueprint for creating objects
   - A class is a new data type that contains member variables and member functions that operate on the variables.
2. **Objects**
   - Objects are the basic run-time entities.
   - They are the instance of a class.
3. **Data abstraction**
   - Abstraction refers to the act of representing essential features without including the back ground details or explanations.
4. **Data Encapsulation**
   - The wrapping up of data and function into a single unit (called class) is known as encapsulation. The data is not accessible to the outside world and only those functions which are wrapped in the class can access it.
5. **Inheritance**
   - Inheritance is the process by which objects of one class acquire the properties of another class. In the concept of inheritance provides the idea of reusability. This mean that we can add additional features to an existing class with out modifying it.

6. **Polymorphism**
    - Polymorphism means the ability to take more than one form. An operation may exhibit different instance. The behavior depends upon the type of data used in the operation.
7. **Dynamic binding**
    - Binding refers to the linking of a procedure call to the code to the executed in response to the call. Dynamic binding means the code associated with a given procedure call is not known until the time of the call at run-time.
8. **Message passing**
    - OOP consists of a set of objects that communicate with each other.
    - A message for an object is a request for execution of a procedure and therefore will invoke a function (procedure) in the receiving object that generates the desired result.

First C++ Program

```cpp
#include <iostream> // Required for input/output operations
int main() {
    std::cout << "Hello, World!" << std::endl; // Print message to the
console
    return 0; // Indicate that the program ended successfully
}
```

Explanation:

1. `#include <iostream>` : This is a preprocessor directive to include the Input/Output stream library.
2. `int main()` : The starting point of the program. Every C++ program must have a `main()` function.
3. The `std` **namespace** exists in the **C++ Standard Library**, which is part of the compiler's implementation of the C++ language.
4. The `::` operator in C++ is called the **scope resolution operator**. It is used to specify the scope of a particular identifier, such as a variable, function, or class, when there are multiple possible scopes.
5. `cout` is an instance of the `ostream` class defined in `std` namespace.
6. `std::cout` : Used to print output to the console.
7. The `<<` operator in C++ is called the **stream insertion operator** when used with **output streams**, such as `std::cout` . It is used to send data to the output stream for display on the console.
8. `"Hello, World!"` : The message that will be displayed on the console.
9. `endl` is a function. However, it behaves like an object when used with `std::cout` because it acts as an argument to the `operator<<` function. Specifically, `operator<<` is **overloaded** to accept manipulators (like `std::endl`).
10. `std::endl` : Ends the current line and flushes the output buffer.
11. `return 0;` : Indicates that the program executed successfully.

How to Run:

1. Save the code in a file with a `.cpp` extension (e.g., `hello_world.cpp`).
2. Compile the program using a C++ compiler

```
g++ hello_world.cpp -o hello_world
```

3. Run the compiled program

```
./hello_world
```

Output

```
Hello, World!
```

What is a namespace ? Why it is Necessary ?

A **namespace** in C++ is a way to group logically related classes, functions, variables, and other identifiers under a unique name. It is used to organize code and avoid naming conflicts in large programs or when using multiple libraries.

Namespaces were introduced in C++ to solve the problem of **name collisions** in large projects. Without namespaces, functions, variables, or classes with the same name in different parts of the program or libraries would cause ambiguity and errors.

Advantages of Namespaces

1. **Prevents Name Collisions**: Avoids ambiguity when different parts of the program or libraries define entities with the same name.
2. **Improves Code Readability**:Groups related entities logically under a meaningful name.
3. **Enhances Modularity**: Encourages splitting large codebases into smaller, independent modules.
4. **Facilitates Library Design**: Makes it easier to integrate multiple libraries into the same project.

Removing the `std` prefix for easy coding

We can directly use Standard Library components without the `std::` prefix, if we give directive to compiler that we are going to use `std` namespace and the risk of name conflicts is on the programmer. The directive used is

```
using namspace std;
```

Hence, the above code will be modified to

```
#include <iostream>
using namespace std;
int main() {
    cout << "Hello, World!" << endl;
```

```
    return 0;
}
```

Standard Input and Output Stream

In **C++**, `cin` and `cout` are the standard input and output streams, respectively, provided by the **Standard Input/Output Library** (`<iostream>`). They are used to perform basic input and output operations in C++ programs.

`cout` (Console Output):

`cout` stands for **"character output"**, used to print data to the console. It uses the **standard output stream** (`stdout`) for display.

- **Purpose**: Used to display/output data to the console (standard output).
- **Syntax**: `cout << expression;`
- The `<<` **operator**, called the "insertion operator," sends the data to the output stream.

Example

```
#include <iostream> // Required for cout and cin
using namespace std;

int main() {
    cout << "Hello, World!" << endl; // Output message
    cout << "The value of 5 + 3 is: " << 5 + 3 << endl; // Output
expression
    return 0;
}
```

Output

```
Hello, World!
The value of 5 + 3 is: 8
```

Chaining Output

You can chain multiple outputs using the `<<` operator. This makes the code concise and expressive.

```
cout << "Name: " << "John Doe" << ", Age: " << 25 << endl;
```

Output

```
Name: John Doe, Age: 25
```

Formatting Output

To improve readability, you can format the output using **manipulators** from the `<iomanip>` library:

- `setw(n)` : Sets the width of the output field.
- `fixed` : Displays floating-point numbers in fixed-point notation.
- `setprecision(n)` : Sets the number of digits after the decimal point.
- `left` / `right` : Aligns the output to the left or right.

```
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
    double price = 123.456;
    cout << fixed << setprecision(2);  // Fixed-point with 2 decimals
    cout << "Price: $" << price << endl;

    cout << setw(10) << left << "Item" << setw(8) << right << "Price" <<
endl;
    cout << setw(10) << left << "Book" << setw(8) << right << "$12.99" <<
endl;

    return 0;
}
```

Output

```
Price: $123.46
Item         Price
Book       $12.99
```

Escape Sequences

Special characters can be displayed using **escape sequences**:

Escape Sequences in C++ (`cout`)

Escape Sequence	Meaning	Example Code	Output
`\n`	Newline	`cout << "Hello\nWorld!";`	Hello World!
`\t`	Horizontal tab	`cout << "Hello\tWorld!";`	Hello World!
`\\`	Backslash	`cout << "This is a backslash: \\";`	This is a backslash: \
`\"`	Double quote	`cout << "He said, \"Hi!\"";`	He said, "Hi!"
`\'`	Single quote	`cout << "It\'s a sunny day.";`	It's a sunny day.
`\a`	Alert (bell sound, may not work on all systems)	`cout << "\a";`	(Bell sound)
`\b`	Backspace (moves cursor back one position)	`cout << "Hello\bWorld!";`	HellWorld!
`\r`	Carriage return (moves cursor to start of line)	`cout << "Hello\rWorld!";`	World!
`\f`	Form feed (moves cursor to next page, rarely used)	`cout << "Hello\fWorld!";`	(May vary by system)
`\v`	Vertical tab	`cout << "Hello\vWorld!";`	Hello World!
`\?`	Question mark	`cout << "Why is this here: \?";`	Why is this here: ?
`\0`	Null character (end of a string)	`cout << "This is\0null";`	This is

Example: Demonstrating Escape Sequences

```cpp
#include <iostream>
using namespace std;

int main() {
    cout << "Newline: Hello\nWorld!\n";
    cout << "Tab: Hello\tWorld!\t2025\n";
    cout << "Backslash: This is \\ a backslash\n";
    cout << "Double Quote: He said, \"Hi!\"\n";
    cout << "Single Quote: It\'s sunny\n";
    cout << "Alert: \a (may not work on all systems)\n";
    cout << "Backspace: Hello\bWorld!\n";
```

```
    cout << "Carriage Return: Hello\rWorld!\n";
    cout << "Form Feed: Hello\fWorld!\n";
    cout << "Vertical Tab: Hello\vWorld!\n";
    cout << "Question Mark: Why is this here \?\n";
    cout << "Null Character: This is\0null\n";
    return 0;
}
```

Output

```
Newline: Hello
World!
Tab: Hello      World!  2025
Backslash: This is \ a backslash
Double Quote: He said, "Hi!"
Single Quote: It's sunny
Alert:  (may not work on all systems)
Backspace: HellWorld!
World!ge Return: Hello
Form Feed: Hello
                World!
Vertical Tab: Hello
                   World!
Question Mark: Why is this here ?
Null Character: This is
```

`cin` (Console Input):

`cin` reads data from the **standard input stream** (`stdin`). It uses the `>>` operator to extract input and stores it in variables.

- **Purpose**: Used to take input from the user via the console (standard input).
- **Syntax**: `cin >> variable;`
- The `>>` **operator**, called the "extraction operator," extracts data from the input stream and stores it in a variable.
- It stops reading input when it encounters whitespace, a newline, or an EOF.

Chaining Input

You can use the `>>` operator multiple times to read multiple inputs in a single line.

```
int a, b, c;
cin >> a >> b >> c;
```

Handling Whitespace

- `cin` stops reading when it encounters whitespace (space, tab, or newline).
- If you want to read a full line (including spaces), use `cin.getline()`.

```
#include <iostream>
using namespace std;

int main() {
    char name[30];
    cout << "Enter your full name: ";
    cin.getline(name, 30);  // Reads the entire line
    cout << "Hello, " << name << "!" << endl;
    return 0;
}
```

Error Handling in `cin`

If the user provides input of the wrong type (e.g., entering a string when an integer is expected), `cin` enters a **fail state** and stops working until the error is cleared. If the user provides invalid input, `cin` enters a fail state. Use the following to handle errors:

- `cin.clear()`: Clears the error flag.
- `cin.ignore(n, delimiter)`: Ignores up to `n` characters until a delimiter is encountered.

```
#include <iostream>
using namespace std;

int main() {
    int age;
    cout << "Enter your age: ";
    cin >> age;

    if (cin.fail()) { // Check if cin is in a fail state
        cout << "Invalid input! Please enter a number." << endl;
        cin.clear();   // Clear the error flag
        cin.ignore(1000, '\n'); // Ignore invalid input
    }
    return 0;
}
```

Output

```
Enter your age: q
Invalid input! Please enter a number.
```

Input and Output Redirection in C++

What is Redirection?

Redirection in C++ allows you to use files or other sources as input/output instead of the default **standard input (keyboard)** and **standard output (console)**.

- **Input Redirection (`<`)**: Reads input from a file instead of the keyboard.
- **Output Redirection (`>`)**: Writes output to a file instead of the console.

How to Use Redirection?

Command-Line Redirection Syntax

1. Input Redirection: `<`
   - Redirects the standard input (`cin`) to read from a file.
   - Example: `./program < input.txt`
2. Output Redirection: `>`
   - Redirects the standard output (`cout`) to write to a file.
   - Example: `./program > output.txt`
3. Append Output: `>>`
   - Appends the output to an existing file (instead of overwriting).
   - Example: `./program >> output.txt`
4. Combined Input and Output:
   - Redirect both input and output together.
   - Example: `./program < input.txt > output.txt`

Example Program

Program: Read from Input, Write to Output

```cpp
#include <iostream>
#include <string>
using namespace std;

int main() {
    string name;
    int age;

    // Take input from user or redirected file
    cin >> name >> age;
```

```
    // Output to console or redirected file
    cout << "Name: " << name << endl;
    cout << "Age: " << age << endl;

    return 0;
}
```

Input File (`input.txt`)

```
John
30
```

Run the Program with Redirection

```
./program < input.txt > output.txt
```

Output File (`output.txt`)

```
Name: John
Age: 30
```

Key Notes on Redirection

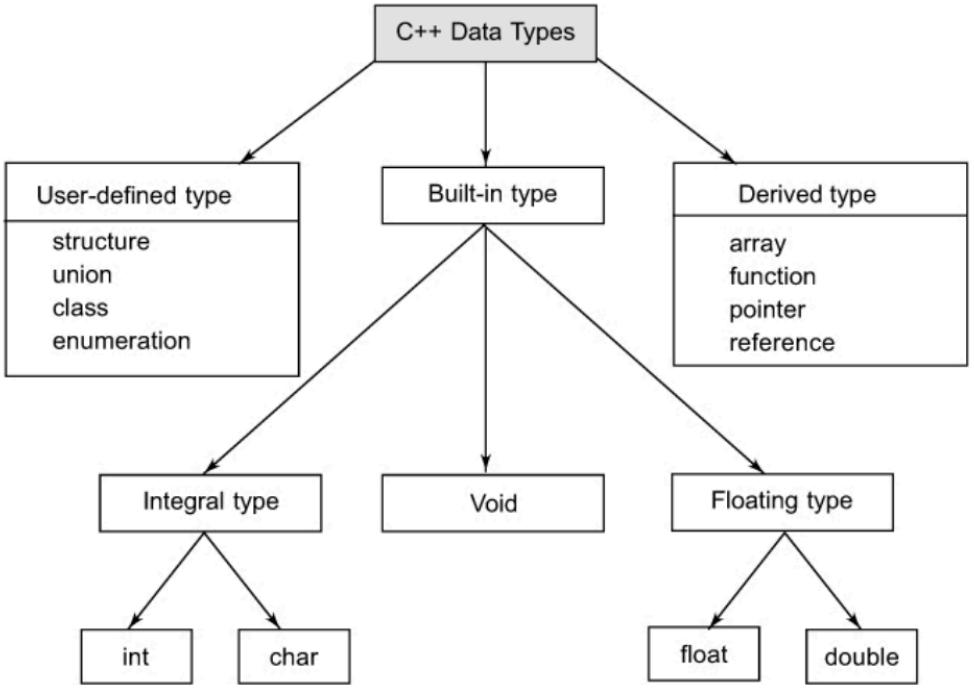
1. **Default Behavior**:
   - `cin` reads from the keyboard.
   - `cout` writes to the console.
2. **With Redirection**:
   - Input (`cin`) reads from the specified input file.
   - Output (`cout`) writes to the specified output file.
3. **Overwrite vs Append**:
   - `>` overwrites the file if it exists.
   - `>>` appends to the file without overwriting.
4. **Error Redirection**:
   - Redirect the standard error (`cerr`) to a file using `2>` .
   - Example: `./program 2> error.txt`

Symbol	Description	Example Command
`<`	Redirect standard input from a file	`./program < input.txt`
`>`	Redirect standard output to a file	`./program > output.txt`
`>>`	Append standard output to a file	`./program >> output.txt`

Symbol	Description	Example Command
`2>`	Redirect standard error to a file	`./program 2> error.txt`
`2>>`	Append standard error to a file	`./program 2>> error.txt`
`<` & `>`	Combine input and output redirection	`./program < input.txt > output.txt`

Datatypes

Data types simply refers to the type and size of data associated with variables and functions.
The Following diagram shows types C++.



C++ data types are broadly categorized into:

1. User-defined Types

Custom types defined by the programmer:

- **struct** – Group of related variables.
- **union** – Memory-efficient structure storing only one member at a time.
- **class** – Object-oriented construct containing data and methods.
- **enum** – Defines a set of named integer constants.

2. Built-in Types

Predefined primitive data types:

- **Integral Type**
  - `int` – Integer numbers
  - `char` – Character data
- **Floating Type**
  - `float` – Single-precision floating-point

- `double` – Double-precision floating-point

- **Void**
  - Represents absence of value, used in functions with no return.

Basic Data Types in C++

a. `int` (Integer)

- Used to store whole numbers.
- Size: Usually 4 bytes (platform-dependent).
- Example:

```
int age = 25;
```

b. `char` (Character)

- Stores a single character.
- Size: 1 byte.
- Stored as ASCII value.
- Example:

```
char grade = 'A';
```

c. `float` (Floating-point number)

- Stores decimal numbers with single precision.
- Size: Usually 4 bytes.
- Example:

```
float temperature = 36.6f;
```

d. `double` (Double-precision floating-point)

- Used for more precision than `float`.
- Size: Usually 8 bytes.
- Example:

```
double pi = 3.1415926535;
```

e. `bool` (Boolean)

- Represents logical values: `true` or `false`.
- Size: Usually 1 byte.
- Example:

```
bool isRaining = false;
```

f. `void`

- Used to indicate **no return type** in functions.
- Cannot define variables of type `void`.
- Example:

```
void displayMessage() {
    cout << "Hello, World!";
}
```

Size and Range of Basic Data Types (Typical on 32/64-bit systems)

Data Type	Size (bytes)	Range
`char`	1	-128 to 127 (signed), 0 to 255 (unsigned)
`bool`	1	`true` or `false`
`int`	4	-2,147,483,648 to 2,147,483,647
`float`	4	~±3.4e±38 (6-7 digits precision)
`double`	8	~±1.7e±308 (15 digits precision)

3. Derived Types

Formed using built-in or user-defined types:

- `array` – Collection of fixed number of elements
- `function` – Reusable block of code
- `pointer` – Holds memory address
- `reference` – Alternative name for a variable

C++ Data Type Modifiers

C++ provides **type modifiers** to alter the **range and storage size** of basic data types (except `void`) to suit different computational needs.

Applicable Data Types

Modifiers can be applied to:

- `int` and `char` (primarily)
- `double` (only `long` is applicable)
- `void` does **not support any modifier**.

List of Modifiers

1. `signed`

- Default for `int` and `char` (unless explicitly stated otherwise)
- Allows both **positive and negative values**
- Example:

```
signed int x = -100;
signed char ch = 'A';
```

2. `unsigned`

- Restricts values to **only positive numbers (including 0)**
- Increases the upper limit of the range
- Example:

```
unsigned int x = 100;
unsigned char ch = 255;
```

3. `short`

- Reduces the **size** of `int` (usually 2 bytes)
- Used when memory saving is critical
- Example:

```
short int x = 32767;
```

4. `long`

- Increases the **storage size** of the data type
- May also be applied to `double` for higher precision
- Examples:

```
long int x = 9223372036854775807;
long double pi = 3.141592653589793238;
```

Combinations of Modifiers with `int`

Modifier Combination	Typical Size	Value Range
`short int`	2 bytes	-32,768 to 32,767
`unsigned short int`	2 bytes	0 to 65,535
`int` / `signed int`	4 bytes	-2,147,483,648 to 2,147,483,647
`unsigned int`	4 bytes	0 to 4,294,967,295
`long int`	4 or 8 bytes	Wider range, platform-dependent
`unsigned long int`	4 or 8 bytes	Wider positive-only range

Note

- Modifiers help fine-tune memory and performance based on program requirements.
- Always consider the platform/compiler as **data type sizes may vary**.

Derived Datatypes

In C++, **derived data types** are data types that are built from the fundamental (basic) data types. They allow us to group basic data or refer to it in more complex ways. Common derived types include **arrays**, **functions**, **pointers**, and **references**. These help manage collections of data, reuse code, and manipulate memory efficiently in C++.

Arrays

An **array** is a collection of elements **of the same type**, stored one after another in memory.

Example:

```
int numbers[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
cout << "The second element is " << numbers[1] << endl;
```

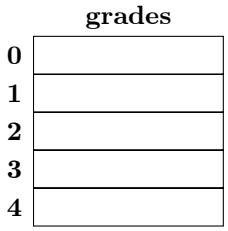
Arrays are data structures that store sets of data items under a single name declaration. Each element of the array must be of the same type. When allocated, an array takes up a contiguous block of memory. The elements can be accessed via an index (a non-negative integer).

Declaring Arrays

An array is declared like any other variable (<type> then <variable name>) but also included is the number of items in the array (the size must be specified when declaring an array). When an array is declared, memory is allocated for the array, and it remains that size. Each element is accessed by an index (or subscript), beginning with the zero-ith element. So, for example, if you need to store a set of 5 integer grades, you can declare an array like the following:

```
int grades[5];
```

where the array will have space for 5 integers indexed from 0-4: grades[0], grades[1], grades[2], grades[3], & grades[4] each one can contain an integer. If you need to access the 3rd item, you would access grades[2] or grades[n-1] where n is the total number of elements in the array. In memory, you can picture it looking something like the following:

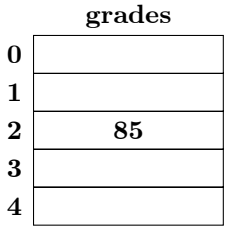


- Once allocated, this allows you to have one variable name for a set of same-type items.
- You can use the entire array as a single object, or each item in the array individually, as shown below.

An individual array element can be used anywhere that a normal variable can be used. For example, if you want to assign the 3rd element the value of 85, then you would write

```
grades[2] = 85;
```

Now our array would look something like this:



Or, if you have an integer variable called `g5` and you want to assign to it the value of the 5th element (assuming it has a value), you would write

```
g5 = grades[4];
```

which then gives `g5` the value of whatever that 5th element is in the array.

Arrays make it much easier for handling large data sets without having to declare each of the items as individual variables. They also allow you to perform subtasks, like filling the elements of an array from user input or a file, displaying all of the elements of an array in forward or reverse order, sorting the values of an array in ascending or descending order, determining the sum, average, or some other statistic from the array values, and finding the highest or lowest value.

With a for loop, for example, you can easily access each item in a large array. Finding the lowest value, or adding up all the values is much easier to implement using an array.

The following loop will sum up all the values in an array of 100 items:

```
for (int i = 0; i< 100; ++i)
    sum += grades[i];
```

Remember to start with zero or you will miss the first item.

TIP: When specifying the size of an array, you can use a named constant or `#defined` value, such as with the following:

```
#include<iostream>
const int MAX_SZ =50;
int main()
{
    int grades[MAX_SZ];
}
```

OR

```
#include<iostream>
#define MAX_SZ 50;
int main()
{
    int grades[MAX_SZ];
}
```

This is helpful for a couple of reasons. First, it improves program readability. It also makes the program more versatile and maintainable because you can change any reference to the size of the array in one easy to find location.

Initializing Arrays

If the initial values of an array are known, the elements in the array can be initialized when declared, such as:

```
int counters[5] = { 0, 0, 0, 1, 5 };
```

The above line of code sets the value of counters [0] to 0, counters[1] to 0, counters[2] to 0, counters[3] to 1, and counters[4] to 5. This would be equivalent to the following 6 lines of code:

```
int counters[5];
counters[0] = 0;
counters[1] = 0;
counters[2] = 0;
```

```
counters[3] = 1;
counters[4] = 5;
```

An array of characters can be initialized in a similar manner:

```
char vowels[5] = { 'a', 'e', 'i', 'o', 'u' };
```

It's not necessary to completely initialize the entire array. If the first few are initialized, the remaining are set to zero (0). Consider the following:

```
float sample_data[500] = { 100.0, 300.0, 500.5 };
```

The first 3 elements are initialized to the values provided; the remaining 497 elements will be set to zero (0). However, no assumptions may be made about the values of the elements of an uninitialized array (i.e. they are NOT zero (0) by default).

If all the elements need to be initialized to something other than zero (0), the best way to do that is to use a for loop.

```
int array_values[10];
int i;
for ( i = 0; i < 10; ++i )
    array_values[i] = i * i;
```

This for loop initializes each element to the square of the element number (subscript number), so the array will contain 0, 1, 4, 9, 16 25, 36, 47, 64, 81 after initialization.

C++ allows you to declare an array without specifying the number of elements only if you initialize every element of the array when it is declared. The following:

```
int counters[] = { 0, 0, 0, 1, 5 };
```

will implicitly dimension the array to 5 elements.

You can also use the index numbers when initializing. So if you know specific elements need to be initialized to a certain value, you can do the following:

```
float sample_data[] = { [0] = 1.0, [49] = 100.0, [99] = 200.0 };
```

Because the largest index number specified above is 99, sample_data will be set to contain 100 elements; the remaining uninitialized elements initialized to zero (0).

Functions

Definition: A **function** is a reusable block of code that performs a specific task.
Example:

```
int add(int x, int y) {
    return x + y;
}
int result = add(3, 4);
cout << "3 + 4 = " << result << endl;
```

Pointers

Definition: A **pointer** is a variable that stores the **memory address** of another variable.
Example:

```
int number = 42;
int *ptr = &number;
cout << "Address stored in ptr: " << ptr << endl;
cout << "Value pointed to by ptr: " << *ptr << endl;
```

References

Definition: A **reference** is an alternate name for an existing variable.
Example:

```
int original = 100;
int &alias = original;
alias = 200;
cout << "original: " << original << endl;
cout << "alias: " << alias << endl;
```

Functions, Call by Reference

What is a Function?

A **function** is a reusable block of code that performs a specific task. Functions help organize code, promote reusability, improve readability, and allow modular programming.

Syntax

```
return_type function_name(parameter_list) {
    // body of function
}
```

Example:

```
int add(int a, int b) {
    return a + b;
}
```

Function Types

- **User-defined functions**: Written by the programmer.
- **Library functions**: Provided by C++ libraries (e.g. `sqrt()`, `abs()`).

Function Declaration vs Definition

Declaration

```
int add(int, int);
```

Tells the compiler a function exists.

Definition

```
int add(int a, int b) {
    return a + b;
}
```

Contains actual code. Declaration not required if defined before use.

Function Parameters

When calling a function in C++, values are passed to parameters. The way values are passed determines whether the function can **modify** the original variable or just work on a **copy**.

Call by Value

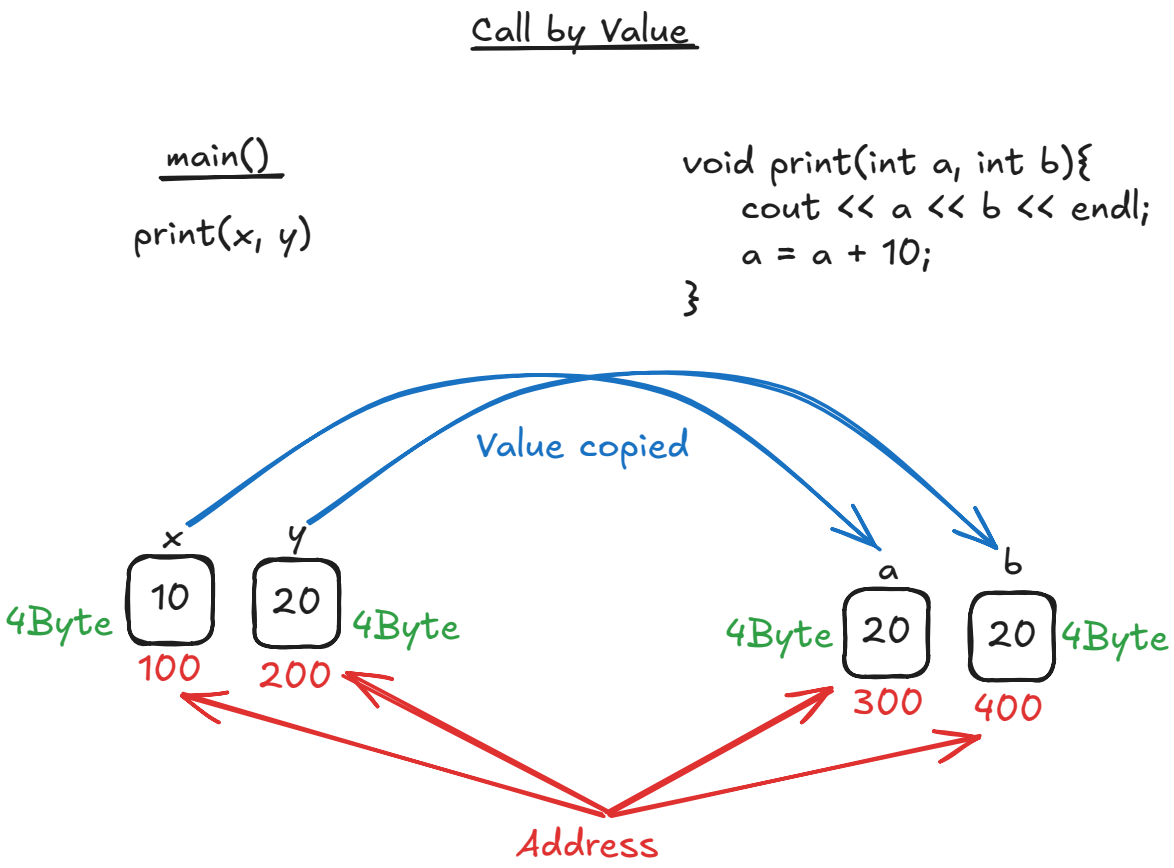
- A **copy** of the actual value is passed.
- Changes made inside the function **do not affect** the original variable.
  Syntax:

```
void modify(int x) {
    x = x + 5;
}
```

Example

```
int main() {
    int a = 10;
    modify(a);
    cout << a;  // Output: 10 (unchanged)
}
```

> **Note**
>
> When original value should remain unchanged.



Call by Address (Using Pointers)

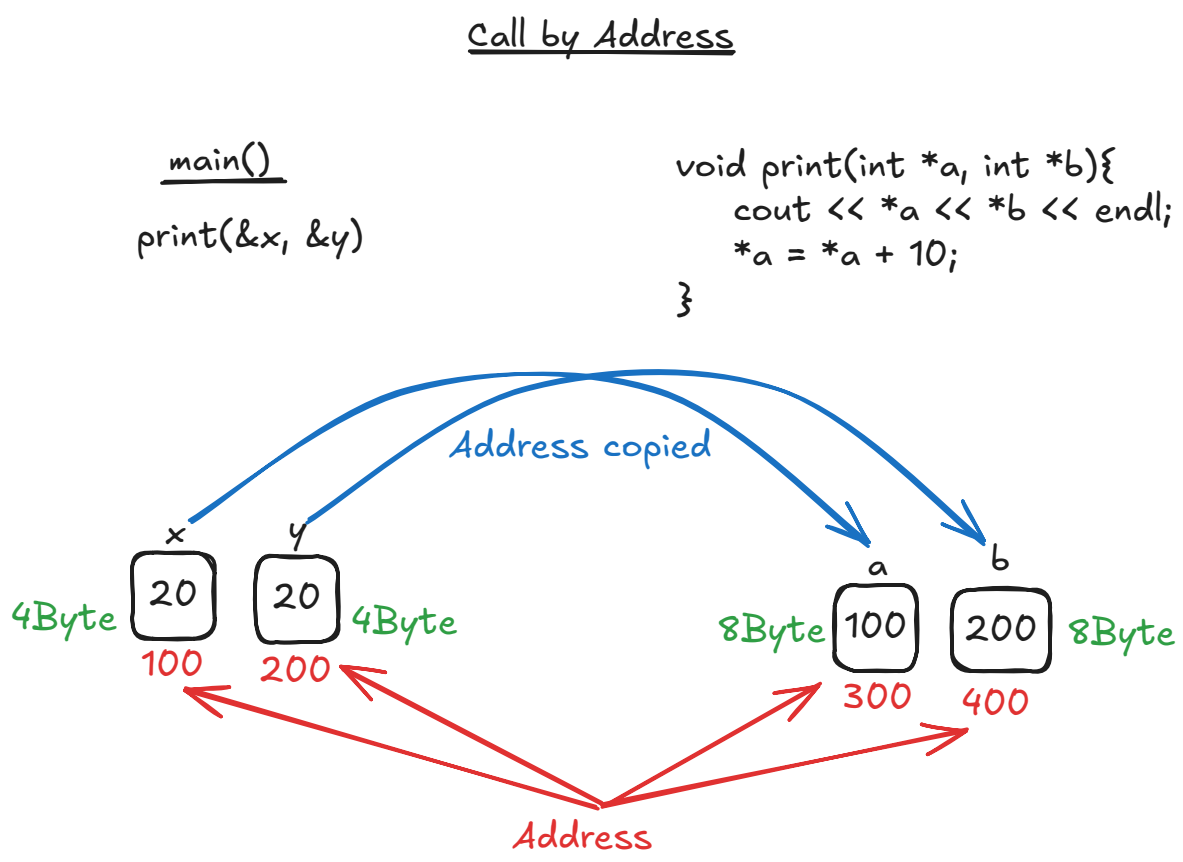
- The function receives the **address** of the variable.
- Modifying the dereferenced pointer changes the original value.
  Syntax

```
void modify(int *x) {
    *x = *x + 5;
}
```

Example

```
int main() {
    int a = 10;
    modify(&a);
    cout << a;  // Output: 15 (modified)
}
```

> **Note**
>
> - When you want to modify the original variable.
> - Used in dynamic memory management and low-level coding.



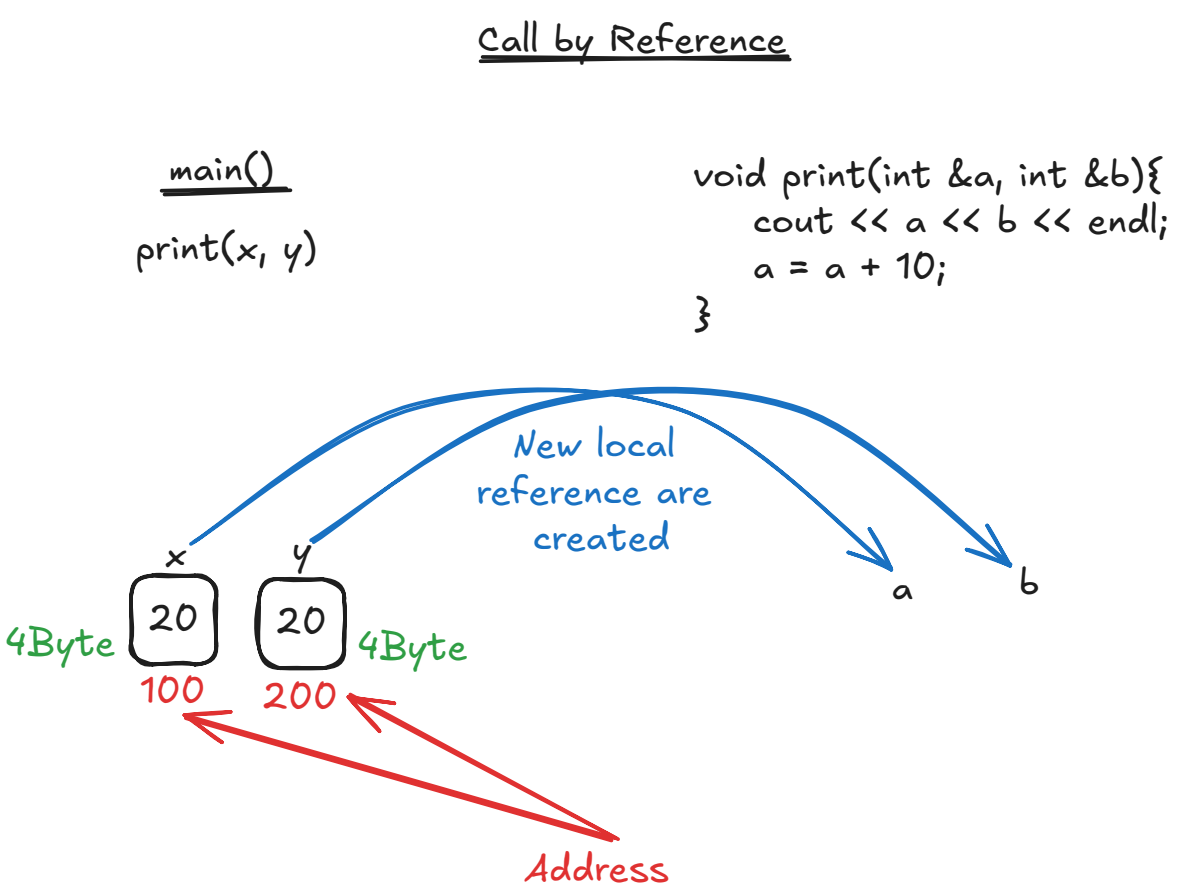
Call by Reference

- A **reference** (alias) of the original variable is passed.
- Changes inside the function affect the original variable **directly**.
  Syntax

```
void modify(int &x) {
    x = x + 5;
}
```

Example

```
int main() {
    int a = 10;
    modify(a);
    cout << a;  // Output: 15 (modified)
}
```



Default Arguments

Allow parameters to have default values.

```
void greet(string name, string prefix = "Mr.") {
    cout << prefix << " " << name;
}
```

Call:

```
greet("John");          // uses "Mr."
greet("Jane", "Dr.");   // uses "Dr."
```

- Must be trailing parameters.
- Declared once, typically in prototype.

Const Arguments

Prevents function from modifying input.

```
void print(const string &msg) {
    cout << msg;
```

```
}
```

- Use with references for safety and performance.
- Enables passing const and temporary objects.

Function Overloading

Function overloading means having **more than one function** with the **same name** in the same scope, but each has **different**:

- Number of parameters
- Type of parameters
- Order of parameters

Syntax

```
int add(int a, int b);           // 2 int parameters
float add(float a, float b);     // 2 float parameters
int add(int a, int b, int c);    // 3 int parameters
```

> **Important**
>
> The correct version is selected based on **arguments passed** during the function call — this process is known as **compile-time polymorphism**.

Example

```
#include <iostream>
using namespace std;

int add(int a, int b) {
    return a + b;
}

float add(float a, float b) {
    return a + b;
}

int add(int a, int b, int c) {
    return a + b + c;
}

int main() {
    cout << add(3, 4) << endl;         // Calls int version
    cout << add(3.5f, 2.5f) << endl;   // Calls float version
```

```
    cout << add(1, 2, 3) << endl;     // Calls 3-parameter version
}
```

Output:

```
7
6
6
```

> **Note**
>
> - Functions must differ in **parameter list**.
> - Return type **alone** cannot distinguish overloaded functions.
>
> **Invalid Example:**
>
> ```
> int show(int a);
> float show(int a);  // ❌ Error: Only return type differs
> ```

Recursion

What is Recursion?

Recursion is a programming technique where a function calls itself to solve a problem. It breaks down a complex problem into smaller instances of the same problem.

Structure of a Recursive Function

Every recursive function must include:

1. **Base Case** – condition to stop recursion.
2. **Recursive Case** – function calls itself with modified parameters.

```
void recursiveFunction() {
    if (base_case_condition)
        return; // base case
    else
        recursiveFunction(); // recursive call
}
```

Example: Factorial

```c
int factorial(int n) {
    if (n <= 1)
        return 1; // base case
    else
        return n * factorial(n - 1); // recursive call
}
```

Example: Fibonacci

```c
int fibonacci(int n) {
    if (n <= 1)
        return n;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Common Issues

- Forgetting the base case (causes infinite recursion).
- Incorrect base case condition.
- Not reducing the problem size.

Class Concept

Class in C++

- In C++, a **class** is a blueprint for creating objects.
- It encapsulates data (variables) and functions (methods) that operate on that data into a single unit.

Key Components of a Class

1. **Data Members**: Variables that hold the data of the class.
2. **Member Functions**: Functions that define the behavior of the class.
3. **Access Specifiers**: Control the access to the class members. Common specifiers:
   - **public**: Members are accessible from outside the class.
   - **private**: Members are accessible only within the class.
   - **protected**: Members are accessible within the class and derived classes.

Class Declaration

A class in C++ is declared using the `class` keyword. It serves as a blueprint for creating objects and defines the properties (data members) and behaviors (member functions).

Syntax:

```
class <ClassName> {
    // Access Specifiers
    // Data Members
    // Member Functions
};
```

Object Creation

Objects are instances of a class. Each object gets its own copy of the class's data members.
Syntax:

```
<ClassName> <ObjectName>;
```

Here, `ClassName` is the name of the class, and `ObjectName` is the name of the object.

Accessing Members

You can access class members using:

- The **dot operator (`.`)** for accessing members of an object.
- The **arrow operator (`->`)** when accessing members through a pointer to the object.

Access Syntax with dot operator

- **Data Members**

```
objectName.dataMember;
```

- **Member Functions**:

```
objectName.memberFunction();
```

Access with Pointers:

- **Data Members**:

```
pointerToObject->dataMember;
```

- **Member Functions**:

```
pointerToObject->memberFunction();
```

Purpose	Syntax Example
Create an object	`ClassName objectName;`
Access data member	`objectName.dataMember;`
Access member function	`objectName.memberFunction();`
Access data member with pointer	`pointerToObject->dataMember;`
Access function with pointer	`pointerToObject->memberFunction();`

Data Members

These are variables that store the attributes of a class. Each object of the class will have its own copy of the data members.
Example

```
class Car {
    string brand; // Data member
```

```
    int speed;    // Data member
};
```

Member Functions

Member functions define the behavior of the class. They operate on the data members of the class.

Syntax:

- Defined **inside the class**:

```
class Car {
    void start() {
        cout << "Car started!" << endl;
    }
};
```

- Defined **outside the class**:

```
class Car {
    void start(); // Function prototype
};

void Car::start() {
    cout << "Car started!" << endl;
}
```

Access Specifiers

Access specifiers control the visibility of class members. There are three types:

- `public` : Members declared as `public` can be accessed from outside the class.
- `private` : Members declared as `private` can only be accessed within the class. This ensures encapsulation and data hiding. The members of a **class** in C++ are **private by default**.
- `protected` : Members declared as `protected` can be accessed in derived (inherited) classes and the class itself.
  Syntax

```
class Car {
private:
    int speed; // Only accessible within the class
```

```
protected:
    int fuel;  // Accessible in the class and derived classes

public:
    string brand; // Accessible from outside the class
};
```

`public`

- Members declared as `public` are **accessible from outside the class**.
- Public members are generally used for interfaces to interact with objects of the class.

Example :

```
#include <iostream>
using namespace std;

class Car {
public:  // Public access specifier
    string brand;  // Public data member
    void display() {  // Public member function
        cout << "Brand: " << brand << endl;
    }
};

int main() {
    Car car1;
    car1.brand = "Toyota";  // Accessing public data member
    car1.display();         // Accessing public member function
    return 0;
}
```

Output

```
Brand: Toyota
```

`private`

- Members declared as `private` are **only accessible within the class**. They cannot be accessed directly from outside the class.
- This is the **default access specifier** if no specifier is provided.
- Private members are typically used to store sensitive or internal data, ensuring **encapsulation**.

```cpp
#include <iostream>
using namespace std;

class Car {
private:  // Private access specifier
    int speed;  // Private data member

public:
    void setSpeed(int s) {  // Public member function to set speed
        if (s > 0)
            speed = s;
        else
            cout << "Invalid speed!" << endl;
    }
    int getSpeed() {  // Public member function to get speed
        return speed;
    }
};

int main() {
    Car car1;
    // car1.speed = 100; // Error: 'speed' is private
    car1.setSpeed(120); // Indirect access through public function
    cout << "Speed: " << car1.getSpeed() << " km/h" << endl;
    return 0;
}
```

`protected`

- Members declared as `protected` are **accessible within the class itself** and **in derived classes**.
- They are not accessible directly from outside the class.
- Protected members are used when designing **inheritance** hierarchies, where child classes can access certain members of the parent class.

Access Specifier	Access Within Class	Access in Derived Class	Access Outside Class	
public	✅ Yes	✅ Yes	✅ Yes	
protected	✅ Yes	✅ Yes	❌ No	
private	✅ Yes	❌ No	❌ No	

Type Definition

What is `typedef` and Why Use It?

- The keyword `typedef` (short for "type definition") in C++ is used to create an alias for an existing data type.
- **Why use `typedef` ?**
  - Using a type alias can make code more readable and easier to maintain.
  - **Simplify complex types:** Long or nested types (like function pointers or templates) become easier to work with by giving them a short name.
  - **Improve code readability:** A well-chosen alias name (e.g. `Pixel` for an `int` representing a screen pixel) makes the code self-documenting and clearer in intent.
  - **Ease maintenance:** Changing the underlying type (for example, for portability or new requirements) can be done by editing the typedef, without touching the rest of the code.
  - **Maintain C compatibility:** Many C libraries and legacy code use `typedef` extensively (e.g. for struct names and pointer types), so understanding it helps in interfacing with such code.

> **NB:**
>
> a `typedef` *does not create a new type*, it simply creates a synonym for an existing type. The alias is treated by the compiler exactly as if you had written the original type

Basic Syntax and Examples

The basic syntax of a typedef declaration is:

```
typedef existing_type AliasName;
```

```
`#include <iostream>
// Define a typedef alias "Integer" for the type int
typedef int Integer;
int main() {
    Integer count = 5;
    // 'count' is an int, via alias
    std::cout << "count = " << count << std::endl;
```

```
    return 0;
}
```

In this example, `Integer` is an alias for `int`. Declaring `Integer count` is exactly the same as declaring `int count`. The compiler treats `Integer` as if it were `int` whenever it sees it.

Using `typedef` with Structures (Structs)

- One common use of `typedef` in C++ is with structures pointer.
- Though C++ treats structure as a datatype we don't need to use struct key word.

```
// C++ style struct (no typedef needed)
struct Point {
    int x;
    int y;
};

Point p1;            // OK in C++ (Point is a usable type name)
```

- How ever incase of pointer or complex pointers we can used `typrdef` to make more sense of datatype.

```
struct Node {
    int data;
    Node* next;
};
typedef struct Node* NodePtr;  // NodePtr is an alias for "pointer to
Node"

Node n;
NodePtr head = &n;  // head is a Node*
```

Now `NodePtr` can be used wherever a pointer to `Node` is needed, simplifying declarations. (We'll discuss pointers more in the next section.)

```
class Rectangle {
public:
    int width, height;
    int area() { return width * height; }
};

// Create a type alias
typedef Rectangle Rect;
```

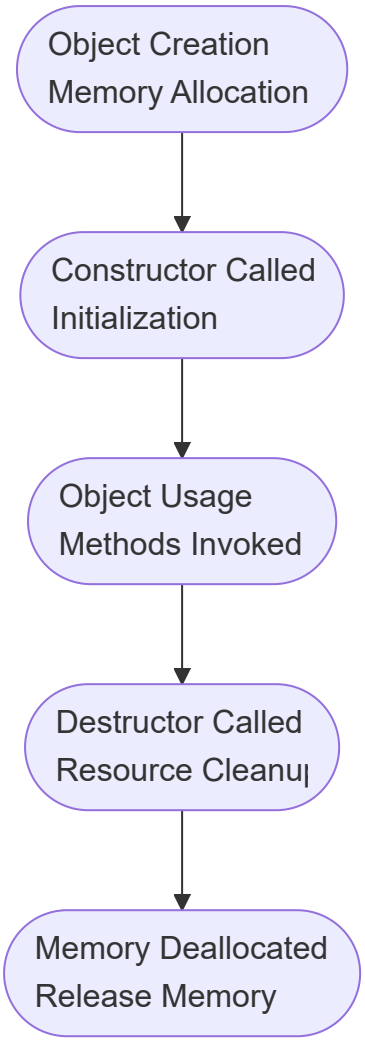
```cpp
int main() {
    Rect r1;
    r1.width = 10;
    r1.height = 5;
    std::cout << "Area: " << r1.area() << std::endl;
}
```

The compiler treats `Rect` exactly the same as `Rectangle`.

Objects, Constructors and Destructors

Object and Constructors

The **object lifecycle** in C++ describes the various stages an object goes through during its existence, from creation to destruction. These stages are defined by how memory is allocated, the constructor is called, the object is used, and eventually destroyed when the destructor is called.



Stages of Object Lifecycle

1. **Object Declaration**:
   - The process of defining an object in code. Memory is allocated for the object when declared.
2. **Constructor Call**:
   - When the object is created, the constructor initializes the object. If no constructor is explicitly defined, the default constructor is used.
3. **Object Usage**:

- The object performs its intended functionality through methods and member functions. During this stage, its state can change.

4. **Destructor Call**:
   - When the object goes out of scope or is explicitly deleted, the destructor is called to clean up resources, releasing allocated memory or performing other final tasks.
5. **Memory Deallocation**:
   - After the destructor finishes execution, the memory allocated to the object is released.

Constructors

- A **constructor** is a special member function in C++ that is automatically called when an object of a class is created. It is primarily used to initialize the object's data members and perform any setup required for the object.

Key Features of a Constructor:

1. **Same Name as the Class:** The constructor's name must match the name of the class.
2. **No Return Type:** Constructors do not have a return type, not even `void`.
3. **Automatically Called:** It is invoked automatically when an object is created.
4. **Can Be Overloaded:** A class can have multiple constructors (constructor overloading) with different parameter lists.

General Syntax of a Constructor

Here's the general syntax for defining a constructor in a class:

```
class ClassName {
public:
    ClassName() {
        // Initialization code
    }
};
```

Types of Constructors

1. Default Constructor

A **default constructor** is a special type of constructor in C++ that **takes no arguments** or **has all default arguments**. It is automatically invoked when an object of a class is created **without passing any arguments**. Its primary purpose is to initialize an object with default values.

Key Characteristics of a Default Constructor

1. **No Parameters**: A default constructor has no parameters or has parameters with default values.
2. **Automatic Invocation**: It is automatically called when an object is created.
3. **Implicitly Defined**: If no constructor is explicitly defined in a class, the compiler provides an implicit default constructor.
4. **User-Defined Option**: A programmer can define a custom default constructor to initialize members with specific values.
5. **Initialization Purpose**: Ensures that the object starts in a valid state.

Example

```
#include <iostream>
using namespace std;
class MyClass {
public:
    MyClass() { // Default constructor
        cout << "Default constructor called!" << endl;
    }
};
int main() {
    MyClass obj; // Default constructor is automatically called
    return 0;
}
```

Implicit Default Constructor

If you don’t define any constructor in your class, the compiler generates a default constructor automatically.

- It does nothing explicitly.
- The data members of built-in types are left uninitialized.
- If a class has no constructors, the compiler automatically generates a default constructor.

```
class MyClass {
    // Compiler provides an implicit default constructor
};
```

Explicit Default Constructor

You can explicitly define a default constructor to control the initialization.

```
class MyClass {
public:
    MyClass() {
        // Initialization code
    }
};
```

Benefits of a Default Constructor

1. **Automatic Initialization**: It ensures that objects start with valid initial values, preventing undefined behavior.
2. **Ease of Use**: Simplifies object creation when no specific initialization is required.
3. **Supports Polymorphism**: Plays a crucial role in initializing base and derived class objects.
4. **Compiler-Generated Simplicity**: Eliminates the need for explicit initialization in simple classes.

When Is a Default Constructor Needed?

1. When you create objects without arguments:

```
MyClass obj;
```

2. When working with **arrays of objects**:

```
MyClass arr[5]; // Requires a default constructor
```

3. When creating **dynamic objects**:

```
MyClass* obj = new MyClass(); // Default constructor is called
```

4. When using **inheritance**:
   - A default constructor ensures proper initialization of base and derived classes.

2. Parameterized Constructor

A **parameterized constructor** is a constructor in C++ that takes arguments. It is used to initialize objects with specific values provided during the object creation, making it more flexible and allowing customized initialization.

Key Features of Parameterized Constructors:

1. **Takes Parameters**:

- Unlike a default constructor, a parameterized constructor accepts arguments to initialize data members.

2. **Called During Object Creation**:
   - When an object is created with arguments, the parameterized constructor is automatically invoked.
3. **Constructor Overloading**:
   - Parameterized constructors can be part of constructor overloading, where multiple constructors exist with different parameter lists.
4. **Initialization**:
   - Typically used to initialize member variables with specific values or perform complex initialization.

Example

```
#include <iostream>
using namespace std;

class MyClass {
    int a, b;
public:
    // Parameterized Constructor
    MyClass(int x, int y) {
        a = x;
        b = y;
    }

    void display() {
        cout << "a = " << a << ", b = " << b << endl;
    }
};

int main() {
    MyClass obj(10, 20); // Parameterized constructor is called
    obj.display();       // Output: a = 10, b = 20
    return 0;
}
```

Advantages of Parameterized Constructor

1. **Flexible Initialization**:
   - Allows different objects of the same class to be initialized with different values.
2. **Avoids Manual Initialization**:
   - Simplifies initialization by encapsulating it within the constructor.
3. **Enhances Code Readability**:

- Using parameterized constructors makes the code more concise and easier to maintain.

🖉 Error Example

```
#include <iostream>
using namespace std;

class MyClass {
    int a, b;
public:
    // Parameterized Constructor
    MyClass(int x, int y) {
        a = x;
        b = y;
    }

    void display() {
        cout << "a = " << a << ", b = " << b << endl;
    }
};

int main() {
    MyClass obj(10, 20); // Parameterized constructor is called
    MyClass obj_err; // Error
    obj.display();       // Output: a = 10, b = 20
    return 0;
}
```

As the user defined Parameterized constructor compiler will not add its default constructor. Hence, user defined default constructor needs to be added also.

```
#include <iostream>
using namespace std;

class MyClass {
    int a, b;
public:
     // Default constructor
    MyClass() {
        a = 0;
        b = 0;
    }
    // Parameterized Constructor
    MyClass(int x, int y) {
        a = x;
```

```
        b = y;
    }

    void display() {
        cout << "a = " << a << ", b = " << b << endl;
    }
};

int main() {
    MyClass obj(10, 20); // Parameterized constructor is called
    MyClass obj_err; // Error
    obj.display();       // Output: a = 10, b = 20
    return 0;
}
```

> **Ambiguity in Constructor Calls:**
>
> - If multiple constructors exist with similar parameter lists, ambiguities may arise. Use distinct parameter types to avoid conflicts.

3. Constructor with Default Arguments

A **constructor with default arguments** is a constructor that allows you to provide **default values** for some or all of its parameters. This means if an object is created without providing certain arguments, the constructor will automatically use the default values.

Key Features

1. **Default Values**:
   - You specify default values in the constructor declaration.
   - If an argument is omitted when creating an object, the default value is used.
2. **Combines Flexibility and Code Simplicity**:
   - Instead of writing multiple overloaded constructors, you can use one constructor with default arguments.
3. **Order of Parameters**:
   - Default arguments must be specified from **right to left**. Non-default arguments should always precede default ones.

Syntax

```
class ClassName {
public:
    ClassName(int a = 0, int b = 0); // Constructor with default
```

```
arguments
};
```

Example

```
#include <iostream>
using namespace std;

class MyClass {
    int a, b;
public:
    MyClass(int x = 0, int y = 0) { // Default arguments
        a = x;
        b = y;
        cout << "Constructor called with a = " << a << " and b = " << b
<< endl;
    }
};

int main() {
    MyClass obj1;         // Uses default values (a = 0, b = 0)
    MyClass obj2(10);     // Uses a = 10, b = 0
    MyClass obj3(10, 20);// Uses a = 10, b = 20
    return 0;
}
```

Destructor in C++

What is a Destructor?

A **destructor** is a special member function in C++ that is called automatically when an object of the class goes out of scope or is explicitly deleted. Its primary purpose is to release resources (like dynamic memory, file handles, etc.) allocated during the object's lifetime.

Key Features of a Destructor

1. **Same Name as the Class**:
   - The destructor has the same name as the class, prefixed with a tilde (`~`).
2. **No Parameters and No Return Type**:
   - A destructor cannot have parameters or a return type.
3. **Automatic Invocation**:
   - It is invoked automatically when the object is destroyed.
4. **One Destructor Per Class**:
   - A class can have **only one destructor**.

5. **Used for Cleanup**:
   - Primarily used to free memory, close files, release resources, etc.

Syntax

```
class ClassName {
public:
    ~ClassName() {
        // Code for cleanup
    }
};
```

Example

```
#include <iostream>
using namespace std;

class MyClass {
public:
    MyClass() {
        cout << "Constructor called!" << endl;
    }

    ~MyClass() {
        cout << "Destructor called!" << endl;
    }
};

int main() {
    MyClass obj; // Constructor is called
    cout << "Object in use..." << endl;
    return 0; // Destructor is called when `obj` goes out of scope
}
```

When is a Destructor Called?

1. When an object goes out of scope:
   - For example, at the end of a function or block.
2. When `delete` is used for a dynamically allocated object:

```
MyClass* obj = new MyClass(); // Constructor called
delete obj; // Destructor called
```

3. When the program ends and global/static objects are destroyed.

Note

- A class can only have **one destructor**.
- Use destructors to release resources and avoid memory leaks.

Enumerated Datatype

1. Enumerated Data Type (`enum`)

Definition:

An **enumerated data type** is a user-defined type consisting of integral constants, giving meaningful names to values.

Syntax:

```
enum EnumName { Value1, Value2, ..., ValueN };
```

Example:

```
enum Color { RED, GREEN, BLUE };
Color c = GREEN;
```

Internal Behavior:

- Each enumerator is assigned an integer value starting from `0` by default.
- `RED = 0`, `GREEN = 1`, `BLUE = 2`
- Custom values can be assigned:

```
enum Color { RED = 1, GREEN = 3, BLUE = 5 };
```

```
#include <iostream>
using namespace std;

enum Day {
    Sunday,
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday
};

int main() {
    Day today = Wednesday;

    if (today == Wednesday) {
        cout << "It's midweek!" << endl;
```

```
    }

    // Print enum values
    cout << "Enum values:" << endl;
    cout << "Sunday = " << Sunday << endl;
    cout << "Monday = " << Monday << endl;
    cout << "Tuesday = " << Tuesday << endl;
    cout << "Wednesday = " << Wednesday << endl;
    cout << "Thursday = " << Thursday << endl;
    cout << "Friday = " << Friday << endl;
    cout << "Saturday = " << Saturday << endl;

    return 0;
}
```

Output :

```
It's midweek!
Enum values:
Sunday = 0
Monday = 1
Tuesday = 2
Wednesday = 3
Thursday = 4
Friday = 5
Saturday = 6
```

Notes

- By default, `enum` assigns **integer values starting from 0**.
- You can customize the values like: `Sunday = 1, Monday = 2, ...`
- `enum` improves code **readability** and **type safety**.

2. Enumerated Class (`enum class`)

Definition:

Introduced in C++11, `enum class` improves type safety and avoids polluting the global namespace.

Syntax:

```
enum class EnumName { Value1, Value2, ..., ValueN };
```

Example:

```
enum class Direction { LEFT, RIGHT, UP, DOWN };
Direction d = Direction::LEFT;
```

```
#include <iostream>
using namespace std;

enum class Color {
    Red = 1,
    Green = 2,
    Blue = 4
};

int main() {
    Color favorite = Color::Green;

    // Output enum value by casting to int
    cout << "Favorite color value: " << static_cast<int>(favorite) <<
endl;

    // Using switch-case with enum class
    switch (favorite) {
        case Color::Red:
            cout << "You chose Red." << endl;
            break;
        case Color::Green:
            cout << "You chose Green." << endl;
            break;
        case Color::Blue:
            cout << "You chose Blue." << endl;
            break;
    }

    return 0;
}
```

Output :

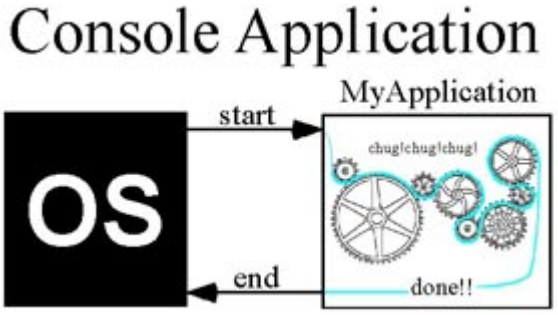
```
Favorite color value: 2
You chose Green.
```

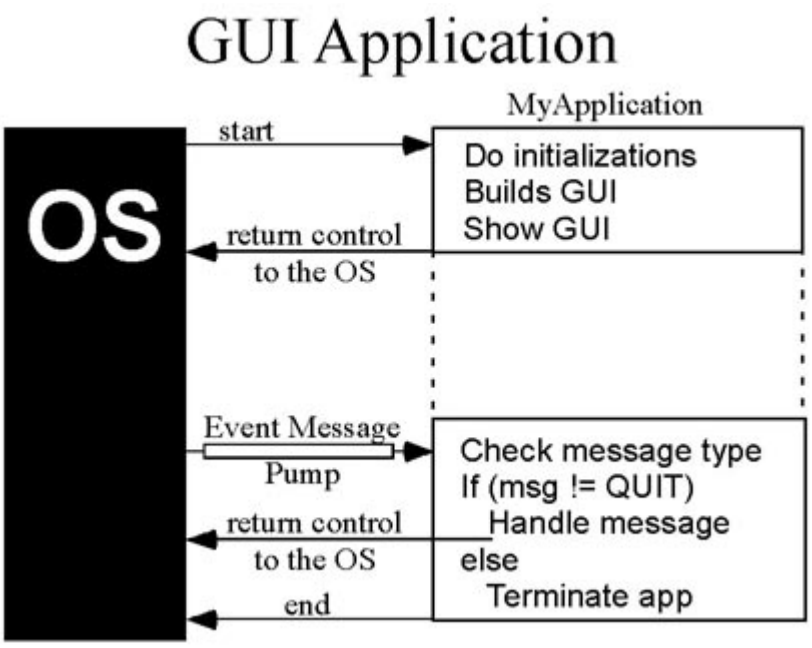
GUI Basics

GUI Basics

1. GUI Programming Basics

- **Different Approach**: Unlike console applications, GUI applications have a distinct execution paradigm.





- **Two Main Parts**: GUI programs consist of
  - Initialization and setup
  - Event handling

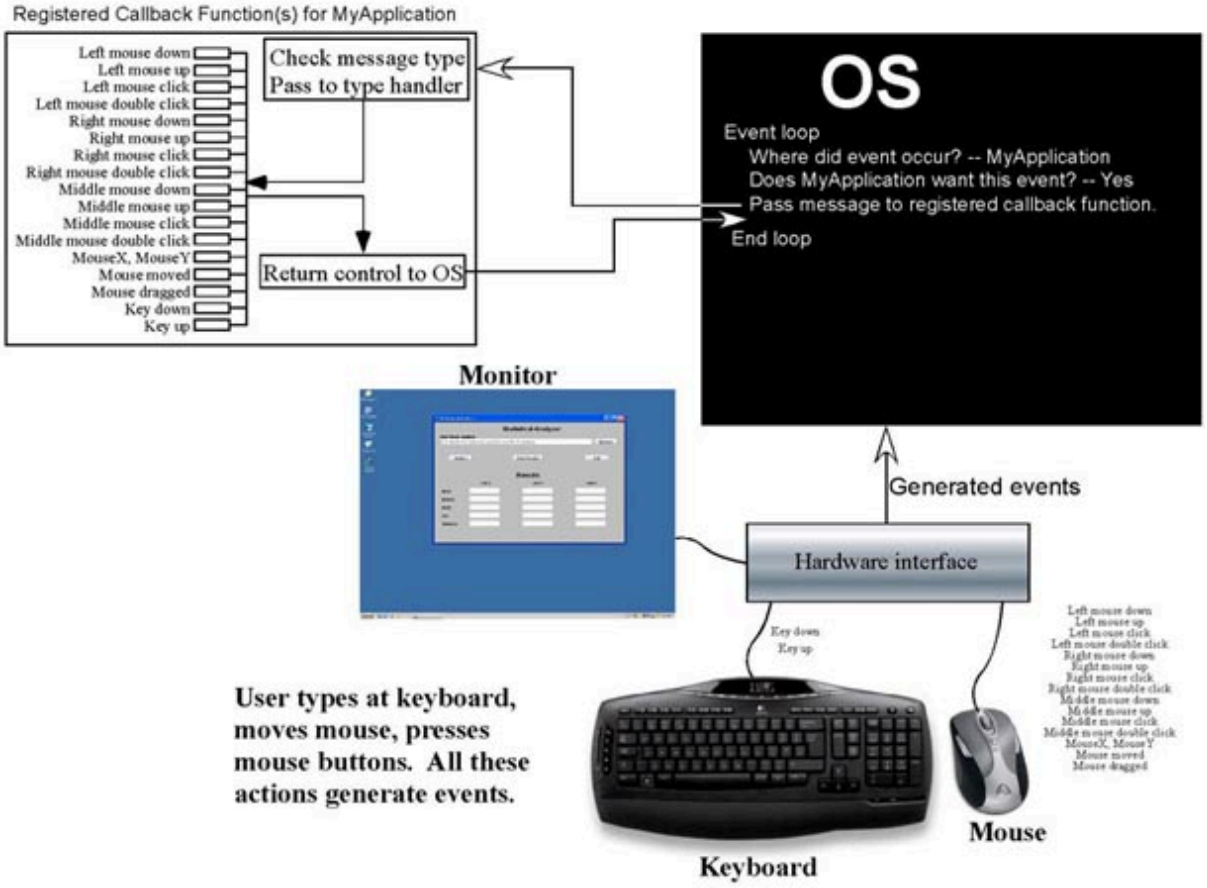
2. Initialization and Setup

- **Data Initialization**: Initialize data structures, instantiate classes, set variable values, etc.
- **Build GUI**: Place widgets (windows, dialog boxes, buttons, etc.) and define the events they handle.
- **Callback Functions**: Define and register callback functions with the operating system.

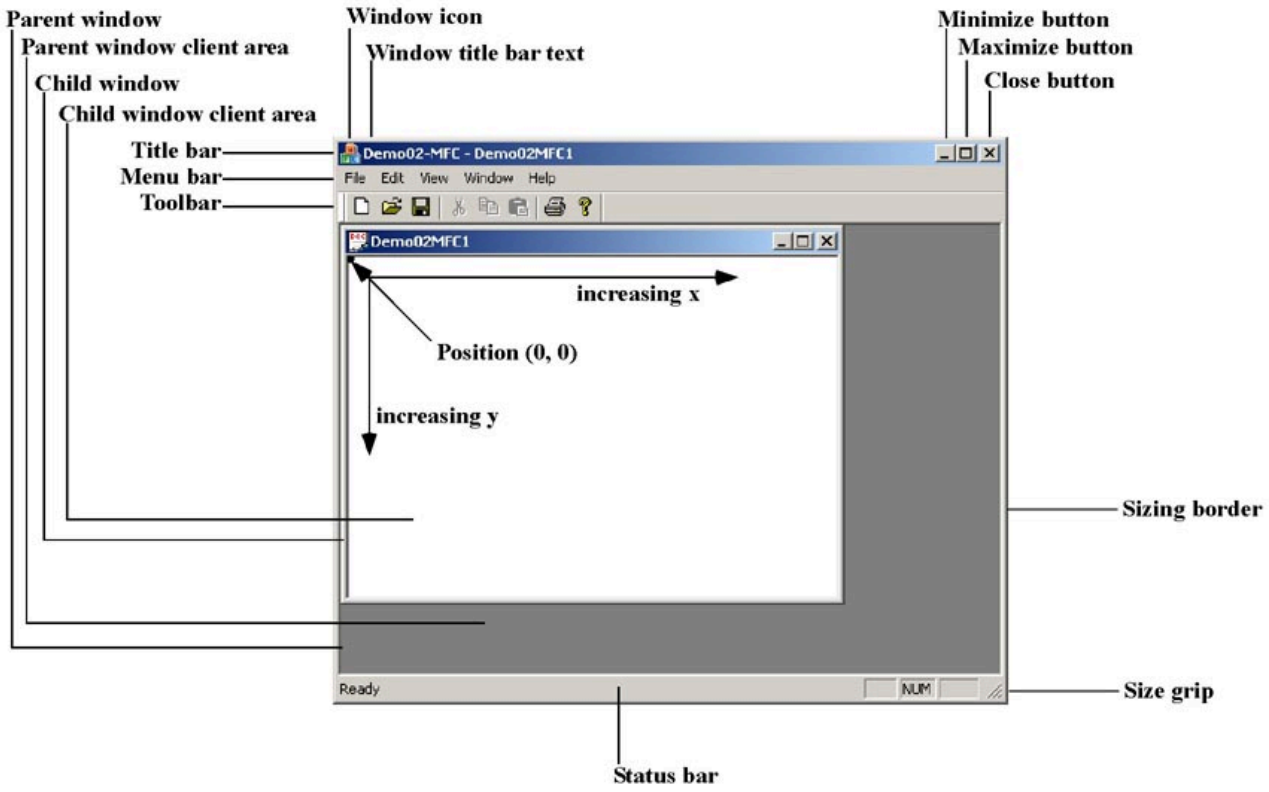
- **Make GUI Visible**: Make the application windows visible and return control to the OS.

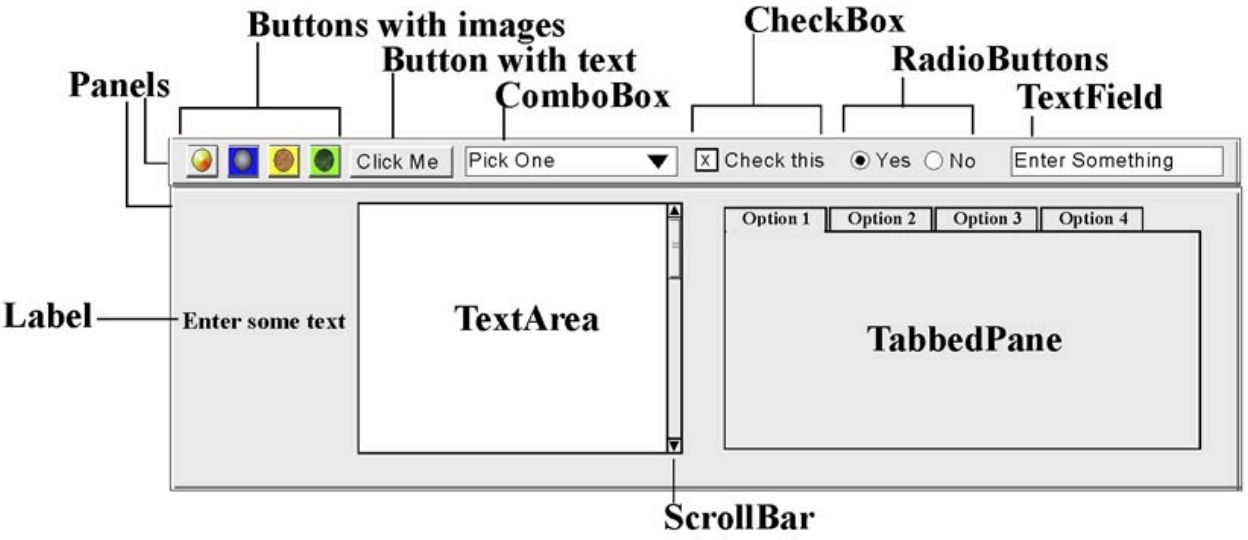
3. Event Handling

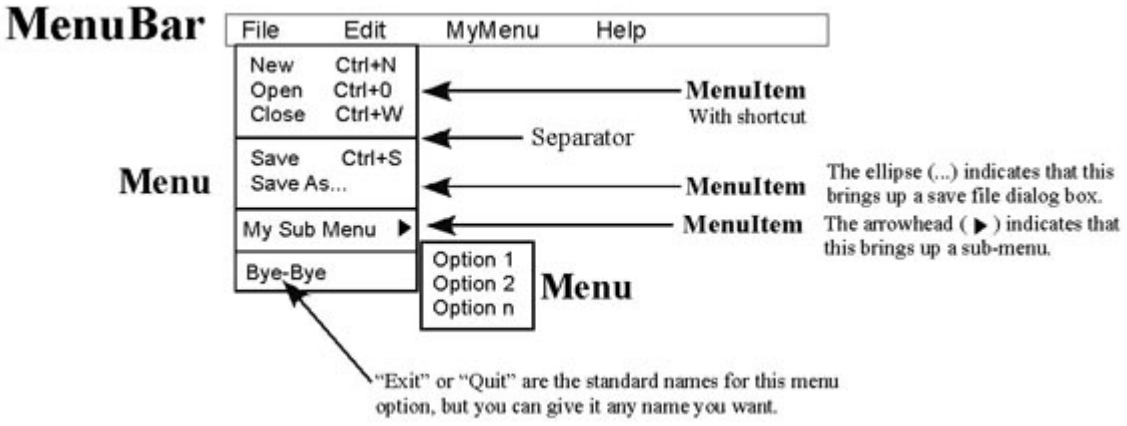
- **Continuous Processing**: After exiting `main()`, the OS continues to process events.
- **Event Messages**: The OS packages events into "event message" structures and passes them to the application.
- **Event-Driven Programming**: Handle events generated by user interactions (mouse clicks, key presses, etc.).



Parts of a GUI



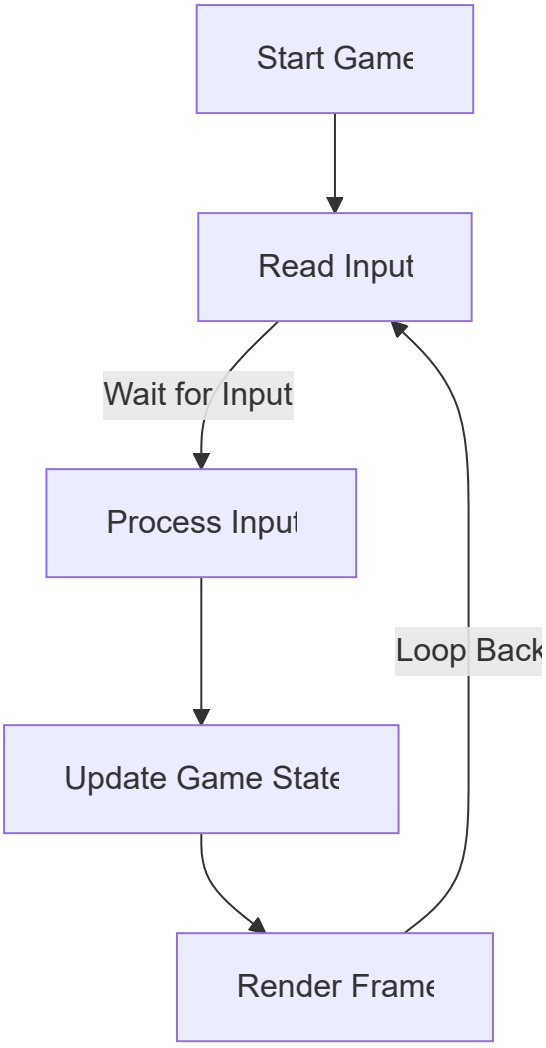




GUI Programs vs Game Programs

Basic Game Loop

- **Continuous Processing**: Unlike event loops, game loops process input continuously and update the game state.
-
- **Three Main Steps**:
  - **process Input**: Handles user input.
  - **update**: Advances the game simulation.
  - **render:** Draws the game.



Timing and Synchronization

- **Frames Per Second (FPS)**: The game loop cycles determine the game's FPS, influencing smoothness.
- **Consistent Speed**: Modern games must run at a consistent speed across different hardware, unlike older games tied to specific hardware.

Game Library

SFML Library

SFML and Linux

- **SFML** (Simple and Fast Multimedia Library) is a free, open-source, cross-platform library designed for multimedia application development. It provides an easy-to-use API for handling various multimedia components, making it a popular choice for game development and other graphical applications.

Features of SFML

- SFML is organized into several modules, each addressing specific aspects of multimedia programming:

1. **System Module**:
   - Provides basic data types and utilities.
   - Handles time, threads, and other low-level functionalities.
2. **Window Module**:
   - Manages window creation, input handling, and OpenGL context settings.
   - Provides an abstraction for keyboard, mouse, and joystick input.
3. **Graphics Module**:
   - Offers 2D graphics rendering capabilities.
   - Supports sprites, shapes, text, and textures.
   - Includes basic transformations, such as scaling and rotation.
4. **Audio Module**:
   - Enables sound and music playback.
   - Supports audio streams and basic sound effects.
   - Works with formats like WAV, OGG, and FLAC.
5. **Network Module**:
   - Provides tools for network communication using TCP and UDP protocols.
   - Handles sockets, packet serialization, and more.

Why Use SFML?

- **Ease of Use**: SFML has a straightforward API that makes it beginner-friendly.
- **Lightweight**: The library is optimized for performance and simplicity.
- **Cross-Platform**: Works on Windows, macOS, Linux, and other platforms.
- **Integration with C++**: Written in C++, SFML takes advantage of modern C++ features.
- **Extensibility**: Can integrate with other libraries, such as OpenGL for advanced rendering.

Typical Applications

- 2D games
- Interactive multimedia applications
- Prototyping and educational projects

Installing the SLFM Library in Linux

If the version of SFML that you want to install is available in the official repository, then install it using your package manager. Use the following command in Debian based system like Ubuntu:

```
sudo apt-get install libsfml-dev
```

Example of a SLFM library code

Assume that the following code is available in prog.cpp

```
#include <SFML/Graphics.hpp>

int main()
{
    sf::RenderWindow window(sf::VideoMode(200, 200), "SFML works!");
    sf::CircleShape shape(100.f);
    shape.setFillColor(sf::Color::Green);
    while (window.isOpen())
    {
        sf::Event event;
        while (window.pollEvent(event))
        {
            if (event.type == sf::Event::Closed)
                window.close();
        }
        window.clear();
        window.draw(shape);
        window.display();
    }
    return 0;
}
```

Compilation Process :

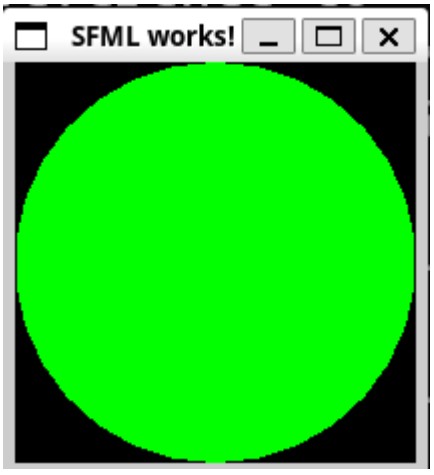
```
g++ prog.cpp -o my-app -lsfml-graphics -lsfml-window -lsfml-system
```

- `g++` is the GNU C++ compiler, which compiles the source code written in C++.
- `prog.cpp` is the source file containing the C++ code to compile.
- `-o my-app` specifies the name of the output executable file as `my-app`.
- `-lsfml-graphics` links the SFML Graphics module to your program. It provides support for rendering 2D graphics like sprites, shapes, and text.
- `-lsfml-window` links the SFML Window module, which handles window creation, OpenGL contexts, and input events such as keyboard and mouse interactions.
- `-lsfml-system` links the SFML System module. It provides utilities like time handling, threading, and other fundamental functions required by other SFML modules.

Running the executable file

```
./my-app
```

Output



Core Modules

The **Core of SFML** consists of five main modules:

1. System Module (`sf::System`)

- This is the lowest-level module and provides utilities such as:
  - `sf::Clock` → Measures elapsed time.
  - `sf::Time` → Represents time durations.
  - `sf::Sleep` → Puts the thread to sleep for a given duration.
  - `sf::Thread` and `sf::Mutex` → Multithreading support.
  - `sf::Vector2<T>` and `sf::Vector3<T>` → Mathematical vector utilities.

2. Window Module (`sf::Window`)

- Handles window creation and user input events.

- Features:
  - `sf::Window` → Creates and manages an application window.
  - `sf::Event` → Captures keyboard, mouse, and joystick inputs.
  - `sf::VideoMode` → Defines resolution, color depth, and refresh rate.
  - `sf::ContextSettings` → Configures OpenGL settings.
  - `sf::Joystick` → Manages game controller inputs.

3. Graphics Module (`sf::Graphics`)

- Provides 2D graphics rendering.
- Features:
  - `sf::RenderWindow` → Extends `sf::Window` with 2D rendering.
  - `sf::Texture` → Handles images.
  - `sf::Sprite` → Displays images efficiently.
  - `sf::Shape`, `sf::RectangleShape`, `sf::CircleShape` → Simple geometric shapes.
  - `sf::Font` and `sf::Text` → Loads fonts and renders text.
  - `sf::View` → Handles camera movement and zooming.

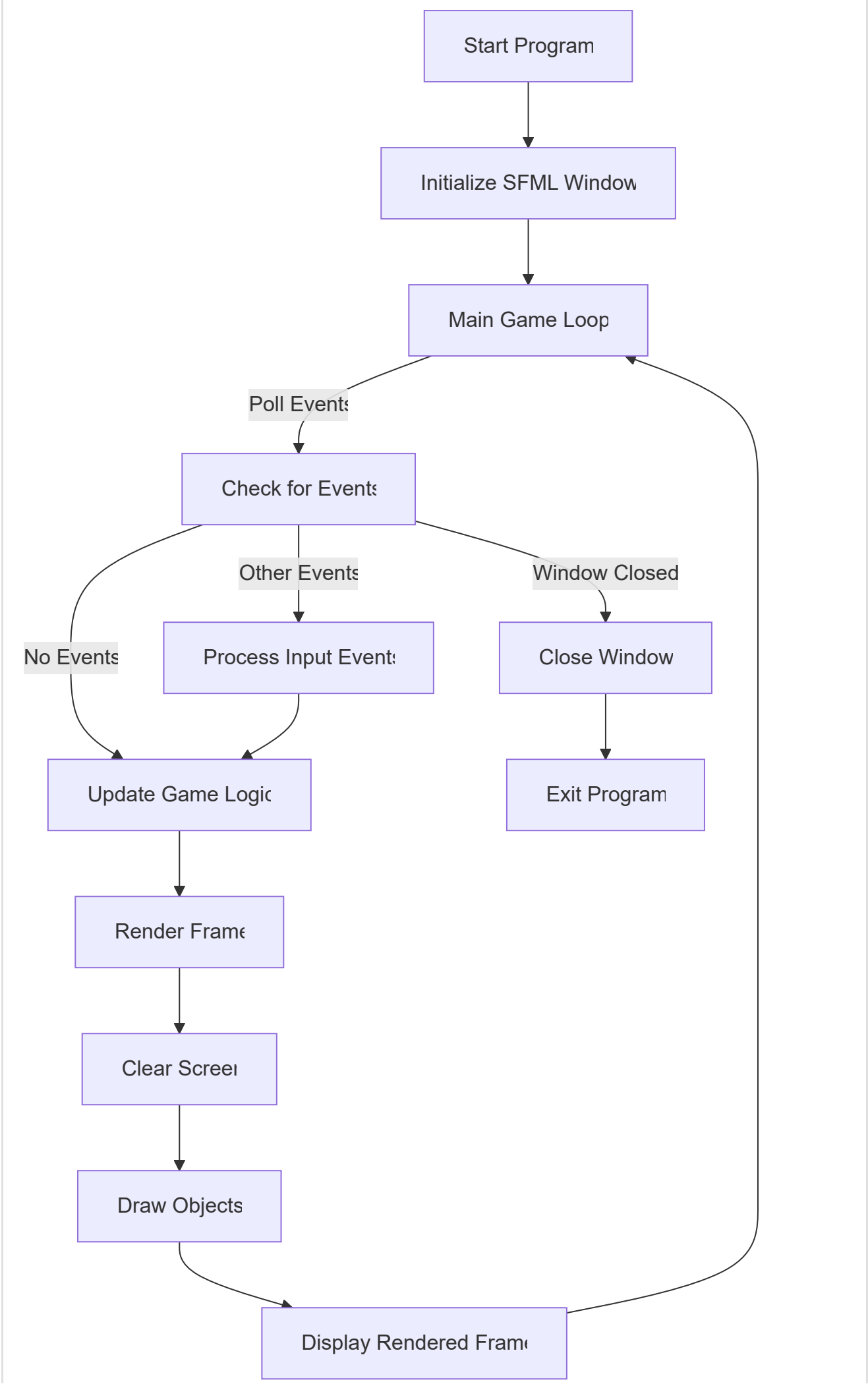
4. Audio Module (`sf::Audio`)

- Provides high-level audio playback.
- Features:
  - `sf::SoundBuffer` → Stores short sound samples.
  - `sf::Sound` → Plays short sounds from `sf::SoundBuffer`.
  - `sf::Music` → Streams large audio files.
  - `sf::Listener` → Simulates 3D audio.
  - `sf::SoundSource` → Base class for sound playback.

5. Network Module (`sf::Network`)

- Provides networking capabilities (TCP and UDP).
- Features:
  - `sf::IpAddress` → Manages IP addresses.
  - `sf::TcpSocket`, `sf::UdpSocket` → Handles network communication.
  - `sf::Packet` → Serializes data for transmission.
  - `sf::Http` and `sf::Ftp` → Allows HTTP and FTP requests.

Stages of SFML Game Program


Start Program
Initialize SFML Window
Main Game Loop
Poll Events
Check for Events
Other Events
Window Closed
No Events
Process Input Events
Close Window
Update Game Logic
Exit Program
Render Frame
Clear Screen
Draw Objects
Display Rendered Frame

Opening a window using SFML

```
// Include important libraries here
#include <SFML/Graphics.hpp>

// Make code easier to type with "using namespace"
using namespace sf;

// This is where our game starts from
int main() {

    // Create a video mode object
    VideoMode vm(1920, 1080);

    // Create and open a window for the game
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

    while(1);
    return 0;
}
```

1. `sf::VideoMode` Class

The `sf::VideoMode` class in SFML represents the video mode of a window, i.e., its **resolution (width & height) and bit depth (color depth in bits per pixel - BPP)**.

```
class sf::VideoMode
```

Common Constructors:

```
sf::VideoMode();
sf::VideoMode(unsigned int width, unsigned int height,
                            unsigned int bitsPerPixel = 32);
```

Example:

```
// Create a 1920x1080 video mode with a default bit depth (32 BPP)
sf::VideoMode vm(1920, 1080);
```

- Here, `1920x1080` is the resolution.
- The bit depth defaults to **32 bits per pixel (BPP)** for full-color rendering.

Important Methods:

```
sf::VideoMode::getDesktopMode();
sf::VideoMode::getFullscreenModes();
```

- `getDesktopMode()` → Gets the current desktop resolution.
- `getFullscreenModes()` → Returns a list of supported fullscreen modes.

Example - Getting Desktop Resolution:

```
sf::VideoMode desktop = sf::VideoMode::getDesktopMode();
std::cout << "Desktop Resolution: " << desktop.width
            << "x" << desktop.height << std::endl;
```

2. `sf::RenderWindow` Class

The `sf::RenderWindow` class is a window that **can render graphics**. It extends `sf::Window` and adds rendering functionalities.

```
class sf::RenderWindow : public sf::Window
```

- `RenderWindow` is a class in SFML used to create a window for rendering graphics.
- The window is created using the `vm` (1920x1080 resolution) and given the title `"Timber!!!"`.
- The **third parameter** (`Style::Fullscreen`) makes the window **full-screen**.
  - Other styles available in SFML:
    - `Style::Default` → Standard window with a title bar and close button.
    - `Style::Close` → A window that can only be closed.
    - `Style::Fullscreen` → Full-screen mode.

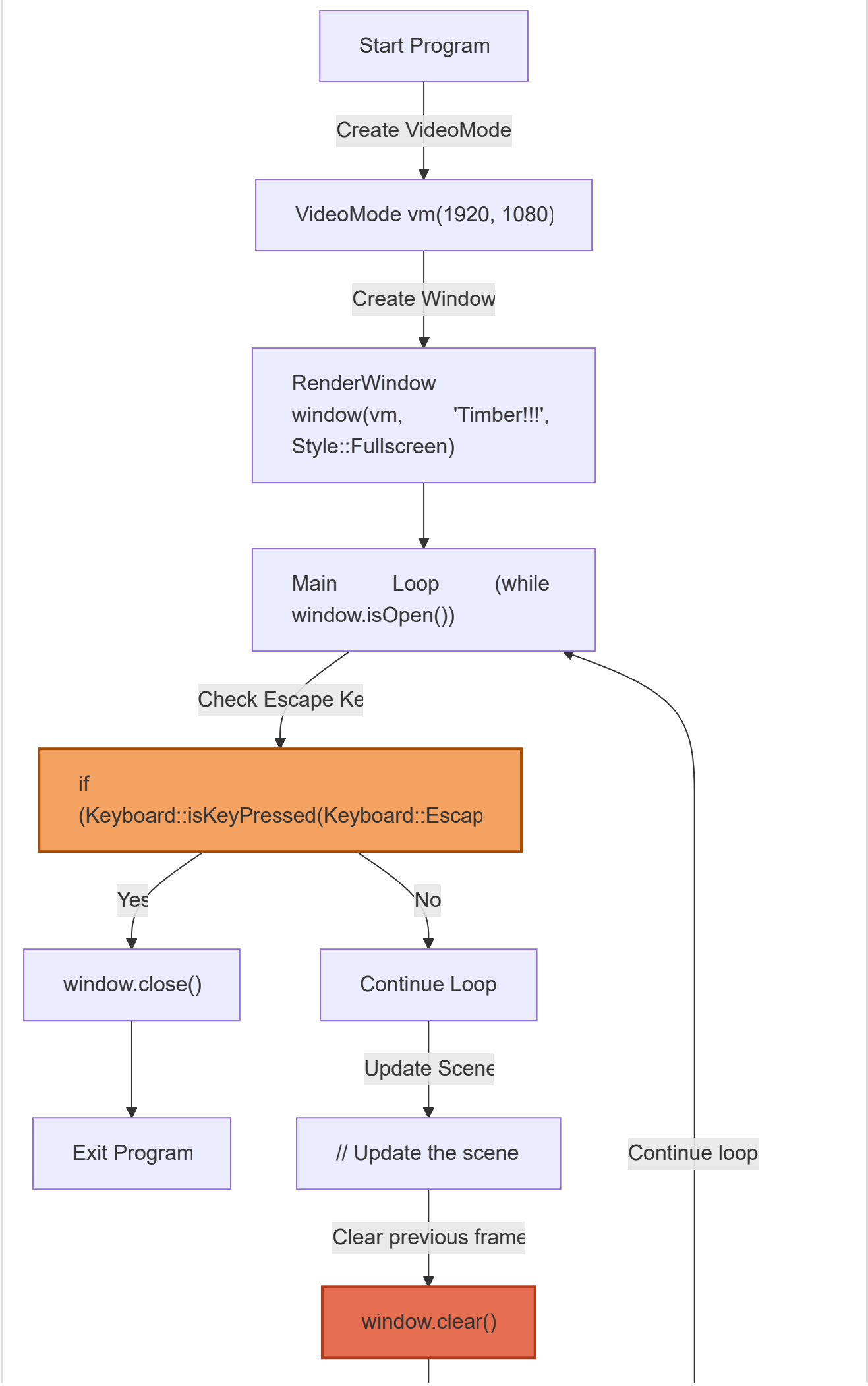
Main Game loop and Use Esc key to exit

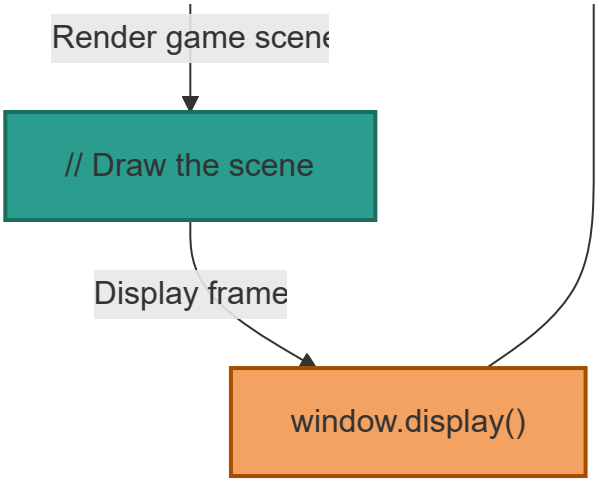
Now we want our game to have a loop where we can design and update the scene as well as set the sequence to exit the full screen mode.

```
#include <SFML/Graphics.hpp>

// Make code easier to type with "using namespace"
using namespace sf;

int main()
{
    VideoMode vm(1920, 1080); // Create a video mode object

    // Create and open a window for the game
```

```
    RenderWindow window(vm, "Timber!!!", Style::Fullscreen);

    while (window.isOpen())
    {
        if (Keyboard::isKeyPressed(Keyboard::Escape)) // Handel player
input
        {
            window.close();
        }

        /*
        Update the scene
        */

        /*
        Draw the scene
        */

        // Clear everything from the last frame
        window.clear();

        // Draw our game scene here

        // Show everything we just drew
        window.display();
    }
    return 0;
}
```

- The function `Keyboard::isKeyPressed(Keyboard::Escape)` is a **static function** provided by SFML (Simple and Fast Multimedia Library) to check whether a specific key is currently being pressed.
- `Keyboard` : This is a class in SFML that provides access to keyboard inputs.
- `isKeyPressed()` : A **static function** that checks whether a key is currently being pressed.
- `Keyboard::Escape` : This refers to the **Escape key** on the keyboard.


Start Program
Create VideoMode
VideoMode vm(1920, 1080)
Create Window
RenderWindow window(vm, 'Timber!!!', Style::Fullscreen)
Main Loop (while window.isOpen())
Check Escape Ke
if (Keyboard::isKeyPressed(Keyboard::Escap
Yes
No
window.close()
Continue Loop
Update Scene
Exit Program
// Update the scene
Continue loop
Clear previous frame
window.clear()



Sprites in a game

A **sprite** is a 2D image or animation used in a game to represent characters, objects, or effects. It is a key element in 2D game development and is typically rendered on the screen as a graphical object.

Key Features of a Sprite:

- **2D Representation** – Unlike 3D models, sprites are flat images displayed on a 2D plane.
- **Moveable & Interactive** – Sprites can be animated, moved, or manipulated in the game.
- **Layered Rendering** – Sprites are drawn on different layers (foreground, midground, background) to create depth.
- **Transparent Background** – Most sprites use PNG images with transparency to blend into the game scene.

Types of Sprites:

1. **Static Sprites** – A single image that does not change (e.g., a tree or rock).
2. **Animated Sprites** – A sequence of images (frames) that create an animation (e.g., walking, jumping).
3. **Sprite Sheets** – A collection of images arranged in a grid to optimize animation and performance.
4. **Tiled Sprites** – Small repeating textures used to create large environments like floors and walls.

How Sprites Work in Games:

- **Rendering:** Sprites are drawn by the game engine onto the screen.
- **Collision Detection:** They interact with other objects based on bounding boxes or pixel detection.
- **Transformation:** They can be moved, rotated, scaled, or flipped based on game logic.

- **Animation:** Frame-by-frame changes give the illusion of movement (e.g., walking cycle).

Where Sprites are Used:

- **2D Games:** Platformers, RPGs, puzzle games (e.g., *Super Mario Bros.*, *Celeste*).
- **UI Elements:** Buttons, icons, menus in both 2D and 3D games.
- **Special Effects:** Fire, smoke, explosions, magic spells.

Types of sprites

1. Character Sprites

- **Player Character (PC)** – The main character controlled by the player.
- **Non-Player Characters (NPCs)** – Characters that are AI-controlled, such as villagers, shopkeepers, or quest givers.
- **Enemies** – Hostile characters that challenge the player, including monsters, soldiers, or bosses.

2. Environment Sprites

- **Background Sprites** – Static or layered images that create the backdrop of the game (e.g., sky, mountains, buildings).
- **Foreground Sprites** – Elements that appear in front of characters, such as trees, fences, or rocks.
- **Platforms and Terrain** – Ground, bridges, floating platforms, and surfaces that characters interact with.

3. Interactive Object Sprites

- **Collectibles** – Coins, gems, power-ups, or keys that the player can collect.
- **Weapons & Tools** – Guns, swords, shields, or tools used by the player or NPCs.
- **Doors & Switches** – Objects that trigger events when interacted with.

4. Animated Effect Sprites

- **Particle Effects** – Fire, smoke, rain, explosions, or magic effects.
- **Hit Effects** – Flashes, damage indicators, or destruction effects.
- **UI Indicators** – Floating damage numbers, health bars, or quest markers.

5. UI and HUD Sprites

- **Health & Mana Bars** – Displays the status of the player’s health and resources.
- **Icons & Buttons** – Inventory icons, ability icons, or in-game buttons.
- **Text and Dialog Boxes** – UI elements that display conversations or information.

6. Parallax Sprites

- **Multiple Background Layers** – Used to create a depth effect where background layers move at different speeds.

How a Game Works: Rendering and Sprite Interaction

A game operates through a continuous **game loop**, which involves:

1. **Processing Input** – Reads player actions (keyboard, mouse, controller).
2. **Updating Game State** – Moves objects, detects collisions, and applies physics.
3. **Rendering Graphics** – Draws game elements on the screen.
4. **Repeating** – Runs at a fixed frame rate (e.g., 60 FPS).

Rendering Process

- **Game Window Setup** – Initializes resolution, aspect ratio, and rendering context.
- **Loading Sprites** – Stores textures in GPU memory for efficient access.
- **Drawing Scene** – Clears the screen and draws objects in layers (background, sprites, UI).
- **Frame Buffer & VSync** – Ensures smooth rendering by managing refresh rates.

Sprite Interaction

- **Movement & Physics** – Sprites respond to user input, velocity, and gravity.
- **Collision Detection** – Detects interactions using bounding boxes, circles, or pixel-perfect methods.
- **Handling Collisions** – Adjusts positions, triggers effects, and executes game logic.

Game Logic & Optimization

- **Game Events** – Manages objectives, health, and AI behavior.
- **Rendering Optimizations** – Uses techniques like batch rendering, Level of Detail (LOD), and culling.
- **State Management** – Handles pause, save/load, and multiplayer synchronization.

Timberman Game

Timberman Game - Objective and Description for Design

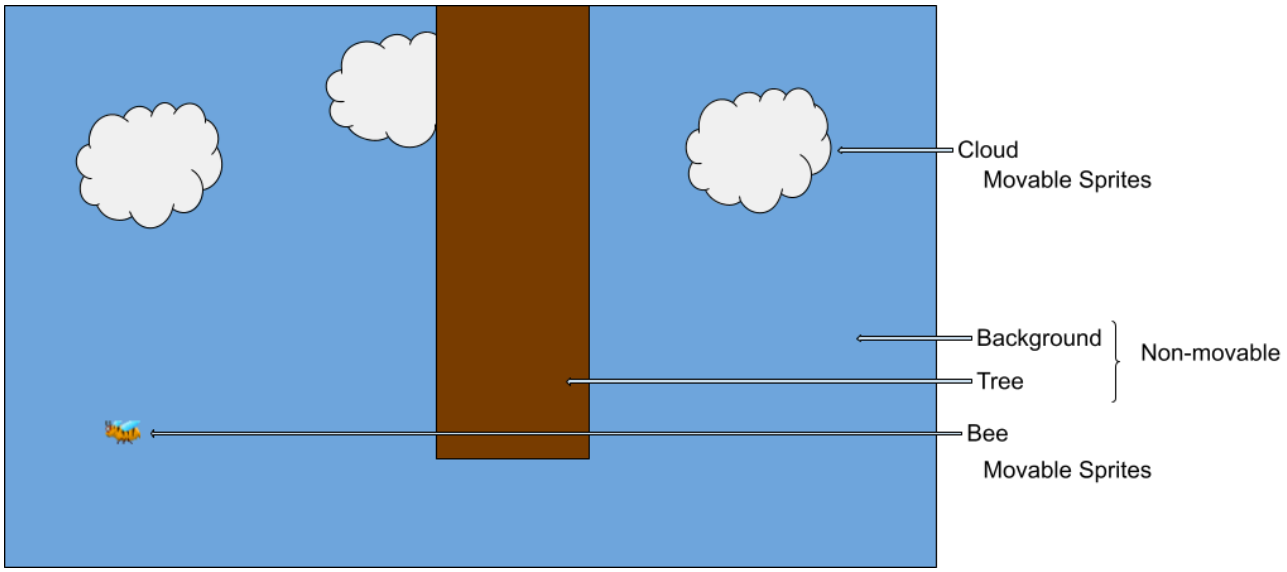
Objective:

The primary goal of **Timberman** is to chop down a tree as fast as possible while avoiding branches. The player must continuously chop to keep the timer from running out. The game ends if the player gets hit by a branch or the timer depletes.

Game Description:

- **Gameplay:** The player controls a lumberjack who stands beside a tall tree. They must chop from either the left or right side while avoiding oncoming branches. The game is fast-paced, requiring quick reflexes and decision-making.
- **Controls:** Typically, the player taps or presses keys to chop the tree on either side.
- **Mechanics:**
  - The tree falls continuously as the player chops.
  - Each chop scores points and slightly refills the timer.
  - Hitting a branch results in an instant game over.
  - The longer the player chops, the faster the tree falls, increasing difficulty.
  - Some versions introduce power-ups or new characters with special abilities.
- **Visual Design:**
  - Simple, pixel-art or cartoonish aesthetic.
  - Background changes with progression.
  - Characters can be customizable or unlockable.
  - Smooth animation for chopping, tree falling, and branch movement.

Consider Environment Sprites




```cpp
//Adding Background
#include<SFML/Graphics.hpp>
using namespace sf;


int main()
{
    VideoMode vm(1920,1080);

    RenderWindow window(vm, "Timber !!!");

    // Create a texture to hold a graphic on the GPU
    Texture textureBackground;

    // Load a graphic into the texture
    textureBackground.loadFromFile("background.png");

    // Create a sprit
    Sprite spriteBackground;

    // Attach the texture to the sprite
    spriteBackground.setTexture(textureBackground);

    // Set the spriteBackground to cover the screen
    spriteBackground.setPosition(0, 0);

    while(window.isOpen())
    {
        if (Keyboard::isKeyPressed(Keyboard::Escape)) // Handel player
input
        {
            window.close();
        }
```

```
            window.clear();

            //Drawing the background sprite
            window.draw(spriteBackground);

            window.display();
        }
    }
```

`sf::RenderWindow` Class

Purpose:

Represents a window where graphical content is displayed.

Member Functions Used:

- **Constructor:** `RenderWindow(VideoMode mode, const std::string& title)`
  - Creates a window with the specified resolution and title.
- `isOpen()`
  - Returns `true` if the window is open, else `false`.
- `close()`
  - Closes the window.
- `clear()`
  - Clears the window and prepares it for new rendering.
- `draw(const Drawable& object)`
  - Draws a drawable object (like `Sprite`) to the window.
- `display()`
  - Displays everything drawn onto the window.

`sf::Texture` Class

Purpose:

Holds an image to be used for rendering.

Member Function Used:

- `loadFromFile(const std::string& filename)`
  - Loads an image file into the texture.

Usage in Code:

```
Texture textureBackground;
textureBackground.loadFromFile("background.png");
```

- Loads the image `"background.png"` into the `textureBackground` object.

`sf::Sprite` Class

Purpose:

Represents a graphical object that can be drawn onto the window.

Member Functions Used:

- `setTexture(const Texture& texture)`
  - Sets a texture for the sprite.
- `setPosition(float x, float y)`
  - Sets the sprite's position in the window.

Understanding the Game coordinates

In 2D games, everything happens on a **flat surface**, like a graph with an X and Y axis.

Coordinate System Basics

- **X-axis**: Horizontal (left ↔ right)
- **Y-axis**: Vertical (up ↕ down)

A position is represented as:
`(x, y)`

For example:

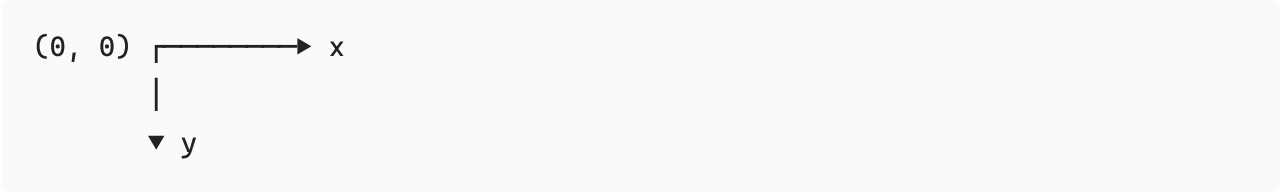
- `(0, 0)` is the **origin** — usually the top-left or bottom-left corner depending on the engine.
- `(100, 200)` means:
  - 100 units **right**
  - 200 units **down** or **up**, depending on the coordinate system

Top-Left vs Bottom-Left Origin

Coordinate System	Origin	Y increases...
Screen-based	Top-left corner	Going **down**
Math-based	Bottom-left corner	Going **up**

Visual Example:

Screen-based (Image coordinate system)



Math-based (Coordinate system)

What Coordinates Are Used For

- Player or object position
- Movement (updating `x`, `y`)
- Collision detection
- Camera tracking
- Mouse/touch input

```
//Add a movable sprite and add movement to it
#include<SFML/Graphics.hpp>
using namespace sf;


int main()
{
    VideoMode vm(1920,1080);

    RenderWindow window(vm, "Timber !!!");

    Texture textureCloud;

    textureCloud.loadFromFile("cloud.png");

    Sprite spriteCloud;

    spriteCloud.setTexture(textureCloud);

    spriteCloud.setPosition(0, 200);

    while(window.isOpen())
    {
        if (Keyboard::isKeyPressed(Keyboard::Escape)) // Handel player
input
        {
```

```
            window.close();
        }

        //Adding movement logic
        spriteCloud.setPosition(spriteCloud.getPosition().x+20,200);

        window.clear();

        //Drawing the cloud sprite
        window.draw(spriteCloud);

        window.display();
    }
}
```

NB: You will see the a cloud moving from left to right. We achieve that by changing the x coordinate of cloud sprite in each frame.

But we don't want the cloud to move out of the window. So as soon as the cloud reaches the right border we want the cloud to restart from left again.

```
//Add a movable sprite and add movement to it
#include<SFML/Graphics.hpp>
using namespace sf;


int main()
{
    VideoMode vm(1920,1080);

    RenderWindow window(vm, "Timber !!!");

    Texture textureCloud;

    textureCloud.loadFromFile("cloud.png");

    Sprite spriteCloud;

    spriteCloud.setTexture(textureCloud);

    spriteCloud.setPosition(0, 200);

    while(window.isOpen())
    {
        if (Keyboard::isKeyPressed(Keyboard::Escape)) // Handel player
input
        {
            window.close();
```

```
        }

        //Adding movement logic
        if(spriteCloud.getPosition().x > 1920)
            spriteCloud.setPosition(0, 200);
        else
            spriteCloud.setPosition(spriteCloud.getPosition().x+20,200);

        window.clear();

        //Drawing the cloud sprite
        window.draw(spriteCloud);

        window.display();
    }
}
```

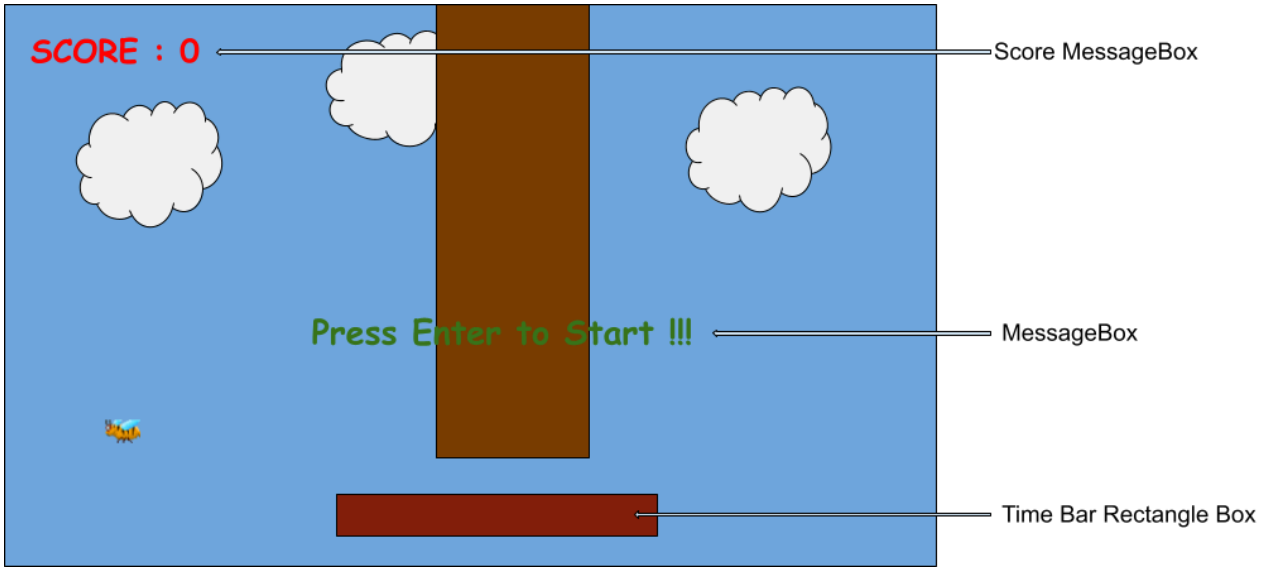
Note

- The movement depends on the frame which is highly unreliable for real time use as frame rendering may not happen with equal time interval.
- Hence, we may want to add the movement to the time elapsed to make sure movement of the sprite according to time interval.

```
//Add a movable sprite and add movement to it

#include<SFML/Graphics.hpp>
using namespace sf;

int main()
{
    VideoMode vm(1920,1080);

    RenderWindow window(vm, "Timber !!!");

    Texture textureCloud;

    textureCloud.loadFromFile("cloud.png");

    Sprite spriteCloud;

    spriteCloud.setTexture(textureCloud);

    spriteCloud.setPosition(0, 200);

    float cloudSpeed = 15; // With how much time it should cover the
```

```
window width
    float cloudSpeedPerSec = 1920/cloudSpeed;

    Time dt; // Object to manipulate the time
    Clock ct; // Object to have a Clock Counter

    while(window.isOpen())
    {
        if (Keyboard::isKeyPressed(Keyboard::Escape))
        {
            window.close();
        }
        dt = ct.restart();

        //Adding movement logic
        if(spriteCloud.getPosition().x > 1920)
            spriteCloud.setPosition(0, 200);
        else
            spriteCloud.setPosition(spriteCloud.getPosition().x+
                                    (dt.asSeconds()*cloudSpeedPerSec),200);

        window.clear();

        //Drawing the cloud sprite
        window.draw(spriteCloud);

        window.display();
    }
}
```

Consider UI and HUD Sprites



What is a HUD?

HUD (Heads-Up Display) refers to on-screen elements that show information like:

- Score
- Health bars
- Ammo count
- Timer
- Minimap

Adding a Score Display (Text UI Element)

To show a score in SFML:

- Use `sf::Font` to load a font.
- Use `sf::Text` to render the score.

Example

```
#include <SFML/Graphics.hpp>
#include <string>

int main() {
    sf::RenderWindow window(sf::VideoMode(800, 600), "Timber !!!");

    // Load font
    sf::Font font;
    font.loadFromFile("KOMIKAP_.ttf");

    // Score Text Setup
    int score = 0;
    sf::Text scoreText;
    scoreText.setFont(font);
    scoreText.setCharacterSize(30);
    scoreText.setFillColor(sf::Color::White);
    scoreText.setPosition(10, 10);
    scoreText.setString("Score: 0");

    // Main loop
    while (window.isOpen()) {
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }

        // Update score
        score += 1;
        scoreText.setString("Score: " + std::to_string(score));
```

```
        // Render
        window.clear();
        window.draw(scoreText);
        window.display();
    }

    return 0;
}
```

Time-Bar HUD

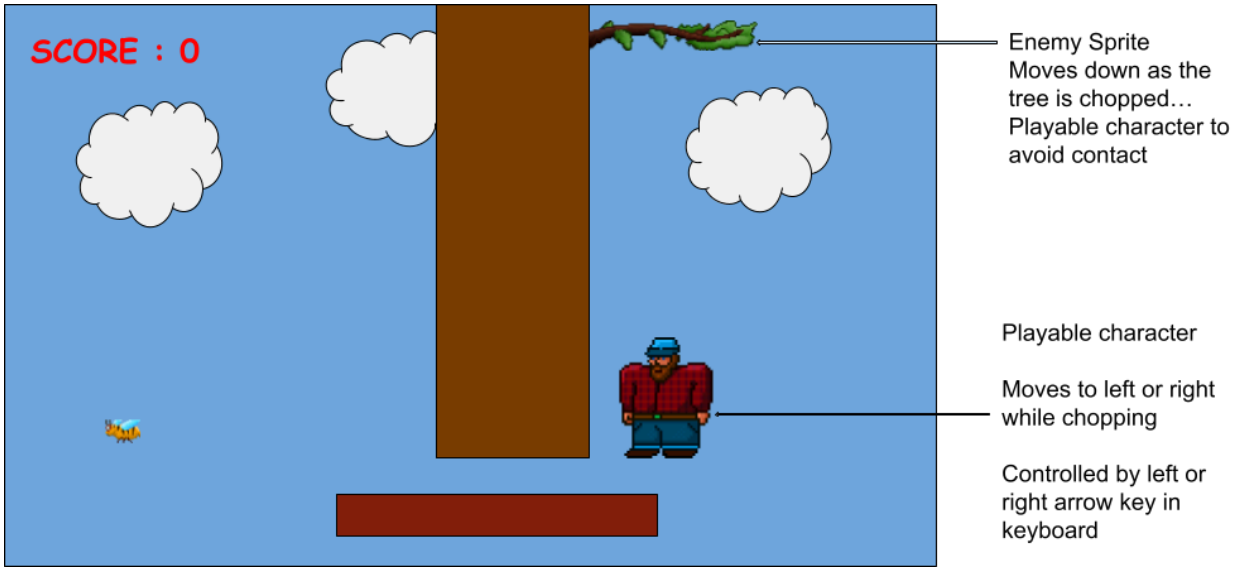
Objective

- Display a time bar (like a health/progress bar) at the **bottom** of the screen.
- The bar **shrinks** over a fixed duration (e.g., 10 seconds).
- After time runs out, the bar is gone (empty).

```
#include <SFML/Graphics.hpp>
using namespace sf;

int main() {
    sf::RenderWindow window(sf::VideoMode(800, 600), "SFML Time Bar");

    // Set total countdown time (in seconds)
    float totalTime = 10.0f;
    float elapsedTime = 0.0f;

    // Initial size of the time bar
    float barWidth = 700.0f;
    float barHeight = 20.0f;

    // Time bar setup
    RectangleShape timeBar;
    timeBar.setSize(sf::Vector2f(barWidth, barHeight));
    timeBar.setFillColor(sf::Color::Red);
    timeBar.setPosition(50, 570); // bottom with margin

    // Clock to track elapsed time
    Clock ct;
    Time dt;

    while (window.isOpen()) {
        dt = ct.restart();
        sf::Event event;
        while (window.pollEvent(event)) {
            if (event.type == sf::Event::Closed)
                window.close();
        }
```

```
        // Update elapsed time
        elapsedTime += dt.asSeconds();

        // Calculate the remaining ratio and update the time bar width
        float remainingRatio = ((totalTime - elapsedTime) / totalTime);
        if(remainingRatio > 0.0f)
            timeBar.setSize(sf::Vector2f(barWidth * remainingRatio,
barHeight));
        else
            timeBar.setSize(sf::Vector2f(0, barHeight));

        // Render everything
        window.clear();
        window.draw(timeBar);
        window.display();
    }

    return 0;
}
```

Consider Character Sprites




```
// TimberMan Game with SFML
// Compile using: g++ TimberMan.cpp -o TimberMan -lsfml-graphics -lsfml-
window -lsfml-system -lsfml-audio

#include <SFML/Graphics.hpp>
#include <SFML/Audio.hpp>
#include <sstream>
#include <cmath>
using namespace sf;

// Updates cloud movement from left to right and resets when off screen
```

```
void cloudUpdate(Sprite &spriteCloud, Time &dt, float
&cloudPixelPerSecond) {
    float x = spriteCloud.getPosition().x;
    float y = spriteCloud.getPosition().y;
    if (x > 1980) {
        spriteCloud.setPosition(-200, y); // reset cloud to left
    } else {
        spriteCloud.setPosition(x + (dt.asSeconds() *
cloudPixelPerSecond), y); // move right
    }
}

// Updates bee movement with sine wave and resets position after off-
screen
void beeUpdate(Sprite &spriteBee, Time &dt, float &beePixelPerSecond) {
    static float timex = 0;
    static float y_dir = 820;
    timex += dt.asSeconds();

    float x = spriteBee.getPosition().x;
    float y = spriteBee.getPosition().y;

    if (x < 0) {
        y_dir = 820 + rand() % 200 - 100; // randomize vertical position
        spriteBee.setPosition(2000, y);   // reset to right
    } else {
        y = y_dir + 50 * sin(timex); // vertical sine motion
        spriteBee.setPosition(x - (dt.asSeconds() * beePixelPerSecond),
y); // move left
    }
}

// Updates the on-screen score text
void scoreUpdate(Text &scoreText, int &score_val) {
    std::stringstream ss;
    ss << "Score = " << score_val;
    scoreText.setString(ss.str());
}

// Updates the time bar; handles time-out scenario
void timebarUpdate(RectangleShape &timebar, float &timeRemaining, bool
&pause, bool &timeOut,
                   Text &messageText, float &timebarHeight, float
&timebarWidthPerSecond, Time &dt, bool logActive) {
    if (!logActive) {
        timeRemaining -= dt.asSeconds();
        if (timeRemaining < 0) {
            pause = true;
            timeOut = true;
            messageText.setString("TimeOut!!!");
```

```
            timebar.setSize(Vector2f(0, timebarHeight)); // hide time bar
        } else {
            if (timeRemaining > 6.0f)
                timeRemaining = 6.0f;
            float newBarWidth = timeRemaining * timebarWidthPerSecond;
            timebar.setSize(Vector2f(newBarWidth, timebarHeight));
        }
    }
}

// Shifts branches downward and spawns new one at the top
void branchUpdate(int &numBranches, Sprite *spriteBranch, float
&chopHeight) {
    for (int i = numBranches - 1; i > 0; i--) {
        float x = spriteBranch[i - 1].getPosition().x;
        float y = spriteBranch[i - 1].getPosition().y;
        if (y > 900) y = -200;
        else y += chopHeight;
        spriteBranch[i].setPosition(x, y);
        spriteBranch[i].setRotation(spriteBranch[i - 1].getRotation());
    }

    // Randomly place a new branch or leave empty
    int r = rand() % 3;
    float x, rotate = 0;
    if (r == 0) { x = 590; rotate = 180; }       // Left
    else if (r == 1) { x = 1330; rotate = 0; }   // Right
    else { x = 3000; }                           // No branch
    spriteBranch[0].setPosition(x, -200);
    spriteBranch[0].setRotation(rotate);
}

int main() {
    VideoMode vm(800, 600);
    RenderWindow window(vm, "Timber Man ");

    View view(FloatRect(0, 0, 1980, 1020));
    window.setView(view);

    // Load and setup background
    Sprite spriteBackground;
    Texture textureBackground;
    textureBackground.loadFromFile("background.png");
    spriteBackground.setTexture(textureBackground);

    // Load and set cloud sprites
    Sprite spriteCloud, spriteCloud2, spriteCloud3;
    Texture textureCloud;
    textureCloud.loadFromFile("cloud.png");
    spriteCloud.setTexture(textureCloud);
```

```
    spriteCloud.setPosition(0, 50);
    spriteCloud.setScale(0.3, 0.3);
    spriteCloud2.setTexture(textureCloud);
    spriteCloud2.setPosition(0, 80);
    spriteCloud2.setScale(0.6, 0.6);
    spriteCloud3.setTexture(textureCloud);
    spriteCloud3.setPosition(0, 120);

    float cloudPixelPerSecond = 1980 / 20.0f;
    float cloud2PixelPerSecond = 1980 / 15.0f;
    float cloud3PixelPerSecond = 1980 / 10.0f;

    // Bee setup
    Sprite spriteBee;
    Texture textureBee;
    textureBee.loadFromFile("bee.png");
    spriteBee.setTexture(textureBee);
    spriteBee.setPosition(1980, 820);
    float beePixelPerSecond = 1980 / 15.0f;

    // Tree
    Sprite spriteTree;
    Texture textureTree;
    textureTree.loadFromFile("tree.png");
    spriteTree.setTexture(textureTree);
    spriteTree.setPosition(810, 0);

    // Score display
    int score_val = 0;
    Text scoreText;
    Font font;
    font.loadFromFile("KOMIKAP_.ttf");
    scoreText.setFont(font);
    scoreText.setString("Score = 0");
    scoreText.setPosition(20, 20);
    scoreText.setCharacterSize(100);
    scoreText.setFillColor(Color::Yellow);

    // Message text (centered)
    Text messageText;
    messageText.setFont(font);
    messageText.setCharacterSize(80);
    messageText.setFillColor(Color::White);
    messageText.setString("Press Enter to start the game !!!!");
    messageText.setPosition(1980 / 2.0, 1020 / 2.0f);
    FloatRect details = messageText.getLocalBounds();
    messageText.setOrigin(details.left + details.width / 2.0f,
details.top + details.height / 2.0f);

    // Time bar setup
```

```
    RectangleShape timebar;
    float timebarHeight = 80;
    float tiembarWidth = 400;
    timebar.setSize(Vector2f(tiembarWidth, timebarHeight));
    timebar.setFillColor(Color::Red);
    timebar.setPosition((1920 / 2.0f) - 200, 1020 - 140);
    float timeRemaining = 6.0f;
    float timebarWidthPerSecond = tiembarWidth / timeRemaining;

    // Branches
    int numBranches = 6;
    Sprite spriteBranch[numBranches];
    Texture textureBranch;
    textureBranch.loadFromFile("branch.png");
    for (int i = 0; i < numBranches; i++) {
        spriteBranch[i].setTexture(textureBranch);
        spriteBranch[i].setPosition(590, -200);
        spriteBranch[i].setOrigin(220, 40);
        spriteBranch[i].setRotation(180);
    }

    float chopHeight = 200;

    // Player setup
    Sprite spritePlayer;
    Texture texturePlayer;
    texturePlayer.loadFromFile("player.png");
    spritePlayer.setTexture(texturePlayer);
    spritePlayer.setPosition(1130, 708);

    // Log setup
    Sprite spriteLog;
    Texture textureLog;
    textureLog.loadFromFile("log.png");
    spriteLog.setTexture(textureLog);
    spriteLog.setPosition(810, 780);
    float speedLog = 0.3;
    float logPixelPexSecX = 990 / speedLog;
    float logPixelPerSecY = 200 / speedLog;
    int logDir = 0;

    // Axe
    Sprite spriteAxe;
    Texture textureAxe;
    textureAxe.loadFromFile("axe.png");
    spriteAxe.setTexture(textureAxe);
    spriteAxe.setPosition(2300, 800);

    // Sound effect
    Sound chop;
```

```
    SoundBuffer chopSound;
    chopSound.loadFromFile("chop.wav");
    chop.setBuffer(chopSound);

    // Rip sprite (death)
    Sprite spriteRip;
    Texture textureRip;
    textureRip.loadFromFile("rip.png");
    spriteRip.setTexture(textureRip);
    spriteRip.setPosition(2300, 750);

    // Game state flags
    bool logActive = false, pause = true, timeOut = false, acceptInput =
true, gamestart = false, gameOver = false;
    Clock ct;
    Time dt;

    // Main game loop
    while (window.isOpen()) {
        dt = ct.restart();

        Event ev;
        while (window.pollEvent(ev)) {
            if (ev.type == Event::Closed) window.close();
            if (ev.type == Event::KeyReleased && !acceptInput) {
                spriteAxe.setPosition(2300, 800);
                acceptInput = true;
            }
        }

        // Input checks and gameplay logic
        if (acceptInput) {
            // Start or restart the game
            if (Keyboard::isKeyPressed(Keyboard::Enter)) {
                if (timeOut || gameOver) {
                    pause = false;
                    timeRemaining = 6.0f;
                    score_val = 0;
                    messageText.setString("");
                    timeOut = false;
                    gameOver = false;

                    // Reset all branches and positions
                    for (int i = 0; i < numBranches; i++)
                        spriteBranch[i].setPosition(3000, -200);

                    spritePlayer.setPosition(1130, 708);
                    spriteRip.setPosition(2300, 750);
                    spriteLog.setPosition(810, 780);
                    logActive = false;
```

```
                spritePlayer.setScale(1, 1);
            } else {
                gamestart = true;
                pause = false;
                messageText.setString("");
            }
            acceptInput = false;
        }

        // Pause the game
        if (Keyboard::isKeyPressed(Keyboard::Space) && gamestart) {
            pause = !pause;
            messageText.setString(pause ? "Game is Paused !!!" : "");
            acceptInput = false;
        }

        // Chop left
        if (Keyboard::isKeyPressed(Keyboard::Left) && !pause) {
            score_val++;
            timeRemaining += (2.0f / score_val) + 0.2f;
            branchUpdate(numBranches, spriteBranch, chopHeight);
            spritePlayer.setPosition(790, 708);
            spriteAxe.setPosition(740, 800);
            spritePlayer.setScale(-1, 1);
            logActive = true;
            logDir = 1;
            chop.play();
            acceptInput = false;
        }

        // Chop right
        if (Keyboard::isKeyPressed(Keyboard::Right) && !pause) {
            score_val++;
            timeRemaining += (2.0f / score_val) + 0.2f;
            branchUpdate(numBranches, spriteBranch, chopHeight);
            spritePlayer.setPosition(1130, 708);
            spritePlayer.setScale(1, 1);
            spriteAxe.setPosition(1040, 800);
            logActive = true;
            logDir = -1;
            chop.play();
            acceptInput = false;
        }
    }

    // Move the log if it's active
    if (logActive && !pause) {
        acceptInput = false;
        float x = spriteLog.getPosition().x;
        float y = spriteLog.getPosition().y;
```

```
            if (x < -100 || x > 2000) {
                // Log goes off screen - reset
                x = 810;
                y = 780;
                logActive = false;
                acceptInput = true;
            } else {
                // Move the log diagonally
                y -= logPixelPerSecY * dt.asSeconds();
                x += logDir * logPixelPexSecX * dt.asSeconds();
            }
            spriteLog.setPosition(x, y);
        }

        // Update game objects if not paused
        if (!pause) {
            cloudUpdate(spriteCloud, dt, cloudPixelPerSecond);
            cloudUpdate(spriteCloud2, dt, cloud2PixelPerSecond);
            cloudUpdate(spriteCloud3, dt, cloud3PixelPerSecond);
            beeUpdate(spriteBee, dt, beePixelPerSecond);
            scoreUpdate(scoreText, score_val);
            timebarUpdate(timebar, timeRemaining, pause, timeOut,
messageText,
                          timebarHeight, timebarWidthPerSecond, dt,
logActive);
        }

        // Game over condition (player hit by branch)
        if
(spritePlayer.getGlobalBounds().intersects(spriteBranch[numBranches -
1].getGlobalBounds())) {
            gameOver = true;
            pause = true;
            score_val = 0;
            messageText.setString("GAME OVER !!!!!");
            timeRemaining = 0;
            timebar.setSize(Vector2f(0, timebarHeight));
            float x = spritePlayer.getPosition().x;
            spriteRip.setPosition(x, 750);
            spritePlayer.setPosition(2300, 708); // hide player
        }

        // Update message text centering
        FloatRect details = messageText.getLocalBounds();
        float originX = details.left + (details.width / 2.0f);
        float originY = details.top + (details.height / 2.0f);
        messageText.setOrigin(originX, originY);

        ////////////////////////////////////////////////
```

```
        // RENDER SECTION

        window.clear(); // clear screen

        // Draw all background and foreground elements
        window.draw(spriteBackground);
        window.draw(spriteCloud);
        window.draw(spriteCloud2);
        window.draw(spriteCloud3);
        window.draw(spriteTree);

        for (int i = 0; i < numBranches; i++)
            window.draw(spriteBranch[i]);

        window.draw(spritePlayer);
        window.draw(spriteLog);
        window.draw(spriteAxe);
        window.draw(spriteRip);
        window.draw(spriteBee);
        window.draw(scoreText);
        window.draw(messageText);
        window.draw(timebar);

        window.display(); // update display
    }

    return 0;
}
```

Pong Game without Class

Game Objectives

1. **Prevent the Ball from Falling**

- Control the **bat/paddle** at the bottom using the **Left** and **Right arrow keys**.
- Ensure the **ball bounces back** by hitting it with the bat.
- If the ball hits the **bottom wall** (misses the bat), the player **loses one life**.

2. **Score Points**

- When the ball hits the **top wall**, the player **earns 1 point**.
- The objective is to **score as many points as possible** before losing all lives.

3. **Survive with Limited Lives**

- The player starts with **3 lives**.
- Each time the ball hits the **bottom** (missed by the bat), **1 life is lost**.
- When all lives are lost:
  - The game shows **"Game Over .. Press Enter to restart"**.
  - The player can press **Enter** to restart the game.

4. **Game Controls**

Action	Key
Move Bat Left	`Left Arrow`
Move Bat Right	`Right Arrow`
Restart After Game Over	`Enter`
Quit Game	`Escape` or Close window

After game over, pressing **Enter** resets:

- Score to `0`
- Lives to `3`
- Ball starts from the **top** with random X direction.

Game Code

```
#include <SFML/Graphics.hpp>
#include <iostream>
#include <sstream>
```

```
using namespace sf;
using namespace std;

int main()
{
    VideoMode video(800, 600);
    RenderWindow window(video, "Pong Game");

    float windowWidth = 1980;
    float windowHeight = 1020;
    View view(FloatRect(0, 0, windowWidth, windowHeight));

    window.setView(view);

    RectangleShape batShape;
    float batWidth = 200;
    float batHeight = 10;
    float batSpeed = 2;
    float batPixSec = windowHeight / batSpeed;
    batShape.setOrigin(batWidth / 2, batHeight / 2);
    batShape.setSize(Vector2f(batWidth, batHeight));
    batShape.setFillColor(Color::White);
    batShape.setPosition(windowWidth / 2, windowHeight - 20);

    CircleShape ballShape;
    float radius = 20;
    float ballSpeed = 3;
    float ballPixSecX = windowWidth / ballSpeed;
    float ballPixSecY = windowHeight / ballSpeed;
    ballShape.setRadius(radius);
    ballShape.setFillColor(Color::Magenta);
    ballShape.setPosition(20 + rand()%(int)(windowWidth-20) , 20);

    Font font;
    font.loadFromFile("KOMIKAP_.ttf");

    Text scoreText;
    scoreText.setFont(font);
    scoreText.setCharacterSize(75);
    scoreText.setFillColor(Color::White);
    scoreText.setPosition(20, 20);

    Text liveText;
    liveText.setFont(font);
    liveText.setCharacterSize(75);
    liveText.setFillColor(Color::White);

    float scoreVal = 0;
    float liveVal = 3;
```

```
    int ballMoveX = rand()%2 == 0 ? -1 : 1;
    int ballMoveY = 1;

    bool paused = false;
    bool bounceTop = false;
    bool bounceBottom = true;
    bool gameOver = false;

    Clock ct;
    Time dt;

    while (window.isOpen())
    {
        dt = ct.restart();
        Event ev;
        while (window.pollEvent(ev))
        {
            if (ev.type == Event::Closed || (ev.type == Event::KeyPressed
&& ev.key.code == Keyboard::Escape))
                window.close();
            if (ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::Enter && gameOver)
            {
                paused = false;
                gameOver = false;
                bounceTop = false;
                bounceBottom = true;
                scoreVal = 0;
                liveVal = 3;
                ballShape.setPosition(windowWidth / 2, 20);
                ballMoveX = rand()%2 == 0 ? -1 : 1;
                ballMoveY = 1;
            }
        }

        stringstream ss, ss1;
        ss << "Score : " << scoreVal;
        scoreText.setString(ss.str());

        if(!gameOver)
            ss1 << "Lives : " << liveVal;
        else
            ss1 << "Game Over .. Press Enter to restart";
        liveText.setString(ss1.str());

        FloatRect details = liveText.getLocalBounds();
        liveText.setPosition(1980 - details.width - 20, 20);

        if (!paused)
        {
```

```
            if (Keyboard::isKeyPressed(Keyboard::Left))
            {
                float x = batShape.getPosition().x;
                float y = batShape.getPosition().y;
                x = x - dt.asSeconds() * batPixSec;
                if (x < batWidth / 2)
                    x = batWidth / 2;
                batShape.setPosition(x, y);
            }
            if (Keyboard::isKeyPressed(Keyboard::Right))
            {
                float x = batShape.getPosition().x;
                float y = batShape.getPosition().y;
                x = x + dt.asSeconds() * batPixSec;
                if (x > windowWidth - batWidth / 2)
                    x = windowWidth - batWidth / 2;
                batShape.setPosition(x, y);
            }
            float x = ballShape.getPosition().x;
            float y = ballShape.getPosition().y;
            x = x + dt.asSeconds() * ballPixSecX * ballMoveX;
            y = y + dt.asSeconds() * ballPixSecY * ballMoveY;
            if (x > windowWidth - radius)
                ballMoveX = -1;
            if (x < radius)
                ballMoveX = 1;
            if ((y > windowHeight - radius ||
batShape.getGlobalBounds().intersects(ballShape.getGlobalBounds())) &&
bounceBottom)
            {
                if (y > windowHeight - radius)
                {
                    liveVal = liveVal - 1;
                }
                ballMoveY = -1;
                bounceTop = true;
                bounceBottom = false;
            }
            if (y < radius && bounceTop)
            {
                ballMoveY = 1;
                scoreVal = scoreVal + 1;
                bounceTop = false;
                bounceBottom = true;
            }
            ballShape.setPosition(x, y);
        }
        if(liveVal == 0)
        {
            paused = true;
```

```
            gameOver = true;
            ballShape.setPosition(windowWidth/2, windowHeight/2);
        }

        window.clear();

        window.draw(batShape);
        window.draw(ballShape);

        window.draw(scoreText);
        window.draw(liveText);

        window.display();
    }
}
```

Pong Game with Class

`Ball.h` File

```
#pragma once
#include<SFML/Graphics.hpp>
using namespace sf;

class Ball{
private:
    CircleShape ballShape;
    float pos_x, pos_y;
    float ballSpeed;
    float ballPixelPerSecX ;
    float ballPixelPerSecy;
    int moveX;//1: Right -1: left;
    int moveY;//1:Down -1:Up
public:
    Ball();
    CircleShape returnShape();
    void upadteBall(Time &dt);
    void ballBounceRight();
    void ballBounceLeft();
    void ballBounceTop(int &score, bool &incScore);
    void ballBounceBottom(bool &incScore, int &live);
    void ballTochesBat(FloatRect batBound, bool &incScore);
};
```

`Ball.cpp` File

```
#include "ball.h"
Ball::Ball()
{
    ballShape.setFillColor(Color::White);
    ballShape.setRadius(40);
    pos_x = 60 + rand()%1870;
    pos_y = 60;
    ballShape.setPosition(pos_x,pos_y);
    moveY= 1;
    moveX = 1;
    ballSpeed = 5;
    ballPixelPerSecX = 1980/ballSpeed;
    ballPixelPerSecy = 1020/ballSpeed;
}
CircleShape Ball::returnShape()
{
```

```
    return(ballShape);
}

void Ball::upadteBall(Time &dt)
{
    float x = ballShape.getPosition().x;
    float y = ballShape.getPosition().y;
     x = x  + moveX * dt.asSeconds()* ballPixelPerSecX;
     y = y + moveY * dt.asSeconds() * ballPixelPerSecy;
     ballShape.setPosition(x, y);
}
void Ball::ballBounceRight()
{
    if(ballShape.getPosition().x > 1900)
        moveX = -1;
}
void Ball::ballBounceLeft()
{
    if(ballShape.getPosition().x < 0)
        moveX = 1;
}
void Ball::ballBounceTop(int &score, bool &incScore)
{
    if(ballShape.getPosition().y < 0 && incScore)
    {
        moveY = 1;
        score = score+1;
        incScore = false;
    }
}
void Ball::ballBounceBottom(bool &incScore, int &live)
{
    if(ballShape.getPosition().y > 940 && !incScore)
    {
        moveY = -1;
        incScore = true;
        live = live -1;
    }
}

void Ball::ballTochesBat(FloatRect batBound, bool &incScore)
{
    if(ballShape.getGlobalBounds().intersects(batBound) && !incScore)
    {
        moveY = -1;
        incScore = true;
    }
}
```

`Bat.h` File

```
#pragma once
#include<SFML/Graphics.hpp>
using namespace sf;
class Bat{
private:
    RectangleShape batShape;
    float batSpeed;
    float batPixelPerSec;
public:
    Bat();
    void moveLeft(Time &dt);
    void moveRight(Time &dt);
    RectangleShape returnShape();
};
```

`Bat.cpp` File

```
#include "bat.h"
Bat::Bat()
{
    batShape.setFillColor(Color::White);
    batShape.setSize(Vector2f(400, 10));
    batShape.setPosition(1980 / 2.0f - 200, 1020 - 40);
    batSpeed = 5;
    batPixelPerSec = 1980 / batSpeed;
}

void Bat::moveLeft(Time &dt)
{
    float x = batShape.getPosition().x;
    float y = batShape.getPosition().y;
    x = x - (batPixelPerSec * dt.asSeconds());
    if (x < 0)
    {
        x = 0;
    }
    batShape.setPosition(x, y);
}

void Bat::moveRight(Time &dt)
{
    float x = batShape.getPosition().x;
    float y = batShape.getPosition().y;
    x = x + (batPixelPerSec * dt.asSeconds());
    if (x > 1580)
    {
```

```
        x = 1580;
    }
    batShape.setPosition(x, y);
}

RectangleShape Bat::returnShape()
{
    return(batShape);
}
```

`PongGame.cpp` File

```
#include <SFML/Graphics.hpp>
#include <sstream>
#include "bat.h"
#include "bat.cpp"
#include "ball.h"
#include "ball.cpp"

using namespace std;
using namespace sf;

int main()
{
    VideoMode vm(800, 600);
    RenderWindow window(vm, "Pong Game!!!");

    View vw(FloatRect(0, 0, 1980, 1020));
    window.setView(vw);

    bool paused = false;
    bool gameOver = false;
    bool acceptInput = true;

    int lives = 3;
    int score = 0;

    Text livesText, scoreText, messageText;
    Font ft;
    ft.loadFromFile("KOMIKAP_.ttf");
    scoreText.setFont(ft);
    scoreText.setFillColor(Color::White);
    scoreText.setCharacterSize(75);
    scoreText.setPosition(20, 20);
    scoreText.setString("Score = 0");

    livesText.setFont(ft);
    livesText.setCharacterSize(75);
```

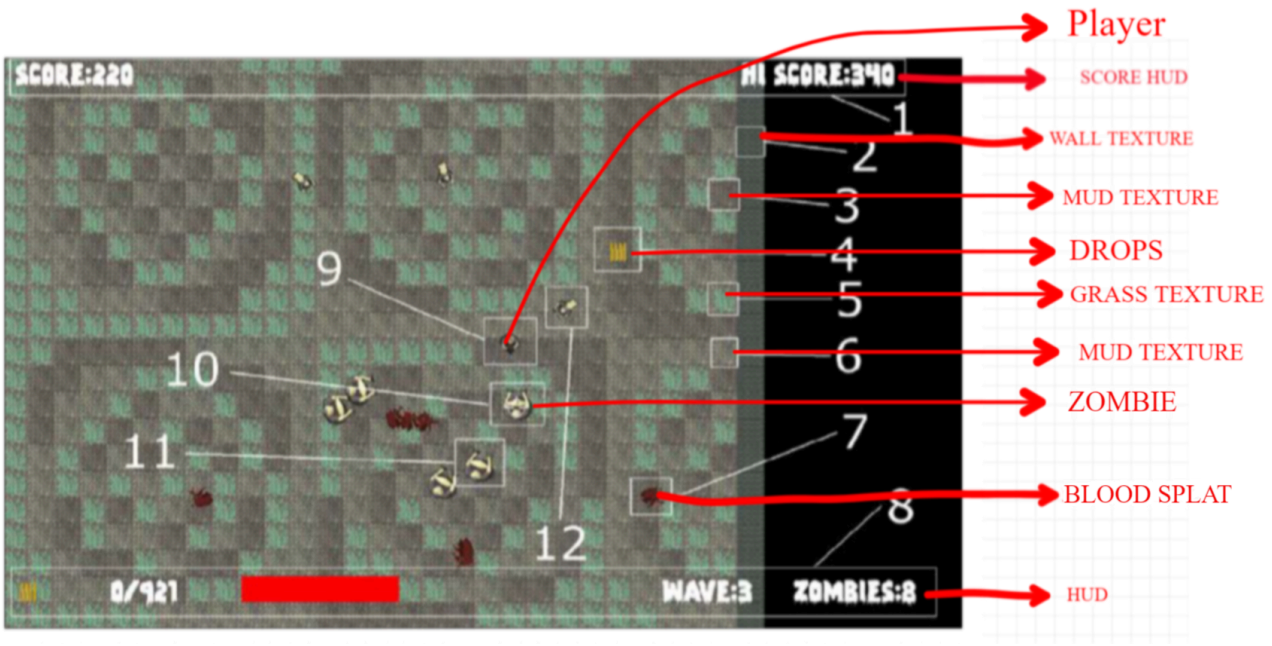
```
    livesText.setFillColor(Color::White);
    livesText.setString("Lives = 3");

    Bat bat;
    Ball ball;

    bool incScore = false;

    Clock ct;
    Time dt;

    while (window.isOpen())
    {
        dt = ct.restart();
        // game Logic
        Event ev;
        while (window.pollEvent(ev))
        {
            if (ev.type == Event::KeyReleased && !acceptInput)
            {
                acceptInput = true;
            }
            if (ev.type == Event::Closed)
            {
                window.close();
            }
        }
        if (acceptInput)
        {
            if (Keyboard::isKeyPressed(Keyboard::Space))
            {
                paused = !paused;
                acceptInput = false;
            }
            if (Keyboard::isKeyPressed(Keyboard::Enter))
            {
                gameOver = false;
                paused = false;
                // re-init
                lives = 3;
                score = 0;
                acceptInput = false;
                incScore = false;
                ball = Ball();
            }
        }
        if (!paused)
        {
            if (lives == 0)
            {
```

```
                gameOver = true;
                paused = true;
            }
            if (Keyboard::isKeyPressed(Keyboard::Left))
            {
                bat.moveLeft(dt);
            }
            if (Keyboard::isKeyPressed(Keyboard::Right))
            {
                bat.moveRight(dt);
            }
            stringstream str_score, str_lives;
            str_score << "Score = " << score;
            str_lives << "Lives = " << lives;
            scoreText.setString(str_score.str());
            livesText.setString(str_lives.str());

            FloatRect boundsLive = livesText.getLocalBounds();
            float y = 20;
            float x = vw.getSize().x - boundsLive.width - 20;
            livesText.setPosition(x, y);

            ball.upadteBall(dt);
            ball.ballBounceRight();
            ball.ballBounceLeft();
            ball.ballBounceTop(score, incScore);
            ball.ballBounceBottom(incScore, lives);
            ball.ballTochesBat(bat.returnShape().getGlobalBounds(),
incScore);
        }
        window.clear();
        // draw the sprites

        window.draw(bat.returnShape());
        window.draw(ball.returnShape());

        window.draw(scoreText);
        window.draw(livesText);
        window.display();
    }
}
```

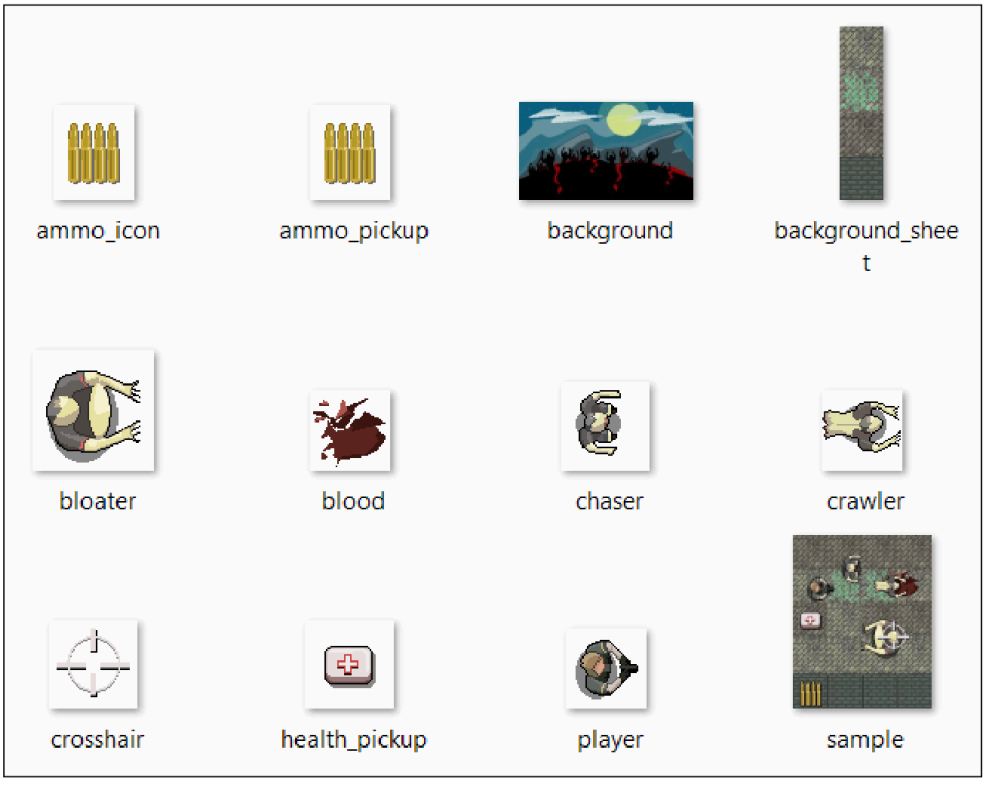
Zombie Arena Game

Primary Objectives

- Learn to draws frames according to user input. (Previously all frame sprites are drawn irrespective of user input.)
- Exploring the SFML View class.



The project assets

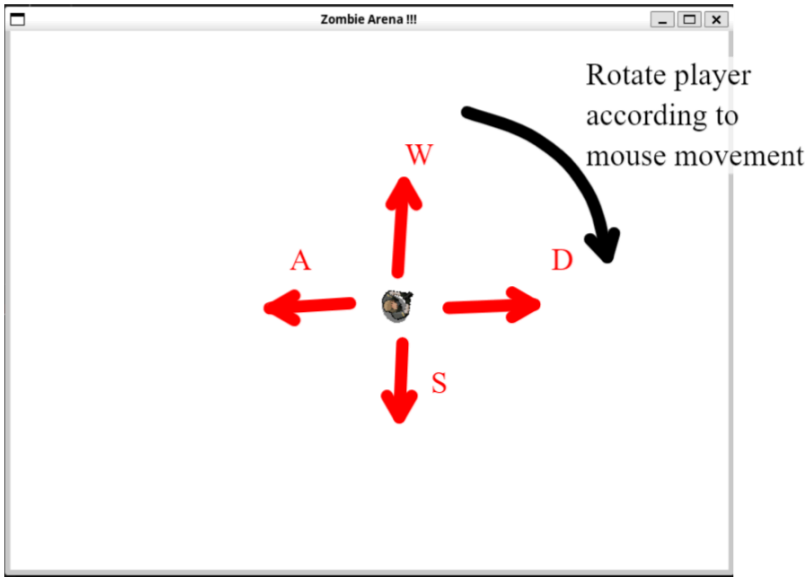


Font : Zombie Control font from http://www.1001freefonts.com/zombie_control.font

Working with W, A, S, D and Mouse

Objective

- Move a player with help of W, A, S, D
- Rotate the player with mouse movement so that it will face towards mouse.

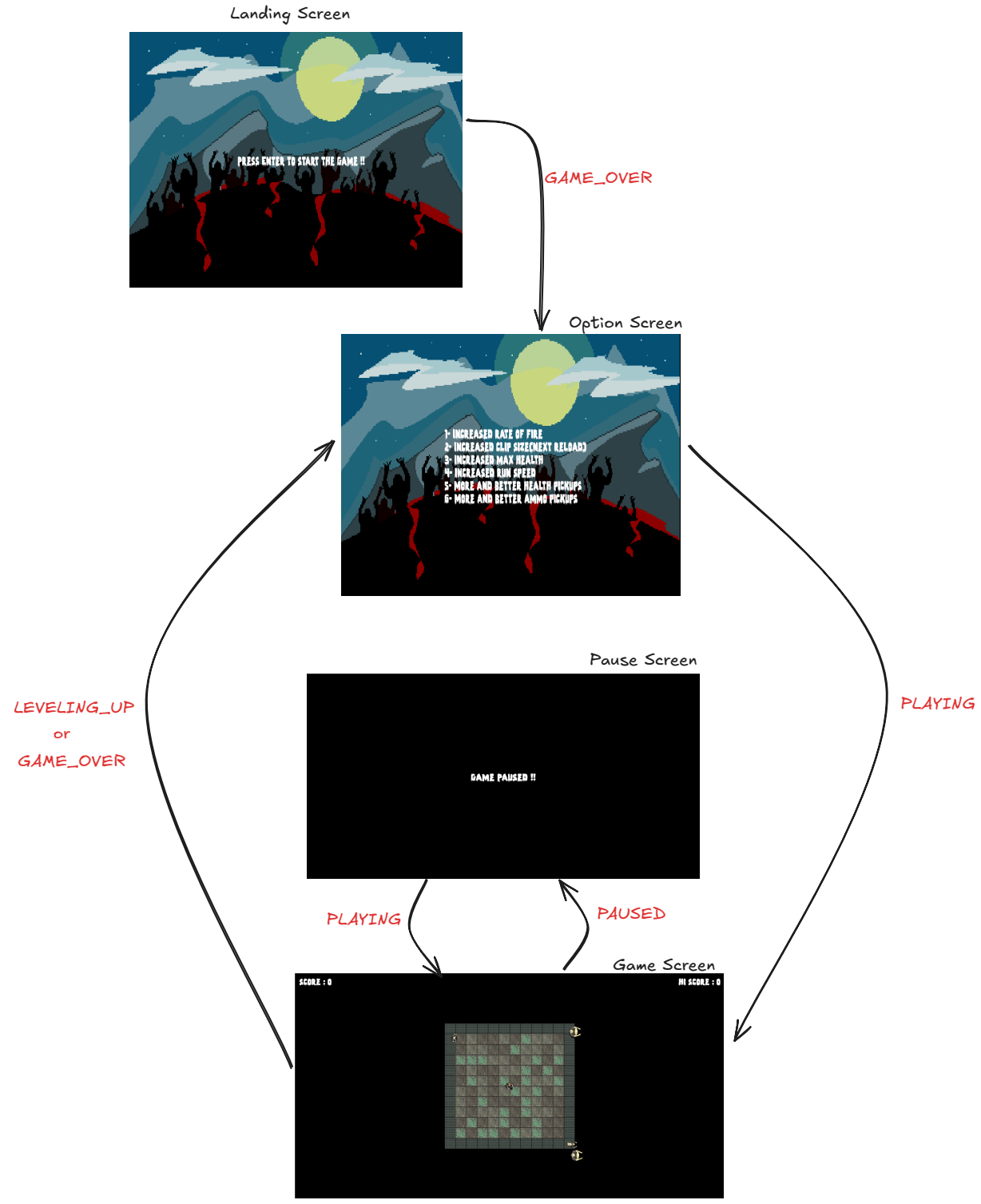



```
#include<SFML/Graphics.hpp>
#include<iostream>
#include<cmath>
using namespace sf;
using namespace std;

int main()
{
    VideoMode video(800, 600);
    RenderWindow window(video, "Zombie Arena !!!");
    Sprite spritePlayer;
    Texture texturePlayer;
    texturePlayer.loadFromFile("player.png");
    spritePlayer.setTexture(texturePlayer);
    FloatRect playerBound = spritePlayer.getLocalBounds();
    spritePlayer.setOrigin(playerBound.width/2, playerBound.height/2);
    float playerSpeed = 5;
    float pixelPerSecPlayerX = window.getSize().x/  playerSpeed;
    float pixelPerSecPlayerY = window.getSize().y/  playerSpeed;
    Clock ct;
    Time dt;
    spritePlayer.setPosition(window.getSize().x/2, window.getSize().y/2);
    while(window.isOpen())
```

```
    {
        dt = ct.restart();
        Event ev;
        while(window.pollEvent(ev))
        {
            if(ev.type == Event::Closed)
            {
                window.close();
            }
        }
        float x = spritePlayer.getPosition().x;
        float y = spritePlayer.getPosition().y;
        if(Keyboard::isKeyPressed(Keyboard::A))
            x = x - dt.asSeconds() * pixelPerSecPlayerX;
        if(Keyboard::isKeyPressed(Keyboard::D))
            x = x + dt.asSeconds() * pixelPerSecPlayerX;
        if(Keyboard::isKeyPressed(Keyboard::W))
            y = y - dt.asSeconds() * pixelPerSecPlayerY;
        if(Keyboard::isKeyPressed(Keyboard::S))
            y = y + dt.asSeconds() * pixelPerSecPlayerY;
        spritePlayer.setPosition(x,y);
        Vector2i mousePos = Mouse::getPosition(window);
        Vector2f mouseWindowPos = window.mapPixelToCoords(mousePos);
        float dx = mouseWindowPos.x - x;
        float dy =mouseWindowPos.y - y;
        float angle = atan2(dy, dx) * (180.0/3.141159f);
        spritePlayer.setRotation(angle);
        window.clear(Color::White);
        window.draw(spritePlayer);
        window.display();
    }
    return(0);
}
```

Game Screens

- In zombie area games there are 4 states:
  - PAUSED
  - LEVELING_UP
  - GAME_OVER
  - PLAYING
- Following Figure Shows all the required Screen based on state of the game




```cpp
#include <SFML/Graphics.hpp>
#include <sstream>
#include <iostream>
#include <cmath>
using namespace sf;
using namespace std;


int main()
{

    enum class State
```

```
    {
        PAUSED,
        LEVELING_UP,
        GAME_OVER,
        PLAYING
    };
    State state = State::GAME_OVER;

    Vector2f windowResolution(800, 600);
    Vector2f gameResolution(1980, 1080);

    VideoMode vm(windowResolution.x, windowResolution.y);
    RenderWindow window(vm, "Zombie Game");

    View gameView(FloatRect(0, 0, gameResolution.x, gameResolution.y));
    window.setView(gameView);

    Sprite spriteBackground;
    Texture textureBackground;
    textureBackground.loadFromFile("background.png");
    spriteBackground.setTexture(textureBackground);

    Font font;
    font.loadFromFile("zombiecontrol.ttf");

    Text textMessage;
    textMessage.setFont(font);
    textMessage.setFillColor(Color::White);
    textMessage.setCharacterSize(50);

    int score = 0;
    Text textScore;
    textScore.setFont(font);
    textScore.setFillColor(Color::White);
    textScore.setCharacterSize(40);
    textScore.setPosition(20, 20);
    textScore.setString("Score : 000");

    int hiScore = 0;
    Text textHiScore;
    textHiScore.setFont(font);
    textHiScore.setFillColor(Color::White);
    textHiScore.setCharacterSize(40);
    textHiScore.setString("Hi Score : 000");
    textHiScore.setPosition(gameResolution.x -
textHiScore.getLocalBounds().width - 20, 20);


    View hudView(FloatRect(0, 0, gameResolution.x, gameResolution.y));
```

```
    while (window.isOpen())
    {
        Event ev;
        while (window.pollEvent(ev))
        {
            if (ev.type == Event::Closed)
            {
                window.close();
            }
            if (ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::Escape)
            {
                window.close();
            }
            if (ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::Enter && state == State::GAME_OVER)
            {
                score = 0;
                state = State::LEVELING_UP;
            }
            if (ev.type == Event::KeyPressed && state ==
State::LEVELING_UP)
            {
                if (ev.key.code == Keyboard::Num0)
                {
                    state = State::PLAYING;
                }
                if (ev.key.code == Keyboard::Num1)
                {
                    state = State::PLAYING;
                }

            }
            if (ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::Space)
            {
                if (state == State::PLAYING)
                {
                    state = State::PAUSED;
                }
                else if (state == State::PAUSED)
                {
                    state = State::PLAYING;
                }
            }
            if (ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::R)
            {
                state = State::GAME_OVER;
```

```
            }
        }

        // Update the Scene
        if (state == State::PAUSED)
        {
            textMessage.setString("Game Paused !!");
        }
        if (state == State::PLAYING)
        {

            textMessage.setString("");
            stringstream scoreStream;
            scoreStream << "Score : " << score;
            textScore.setString(scoreStream.str());

            stringstream hiScoreStream;
            hiScoreStream << "Hi Score : " << hiScore;
            textHiScore.setString(hiScoreStream.str());
            textHiScore.setPosition(gameResolution.x -
textHiScore.getLocalBounds().width - 20, 20);
        }
        if (state == State::LEVELING_UP)
        {
            std::stringstream levelUpStream;
            levelUpStream << "1- Increased rate of fire"
                          << "\n2- Increased clip size(next reload)"
                          << "\n3- Increased max health"
                          << "\n4- Increased run speed"
                          << "\n5- More and better health pickups"
                          << "\n6- More and better ammo pickups";
            textMessage.setString(levelUpStream.str());
            textMessage.setPosition(gameResolution.x / 2 -
textMessage.getLocalBounds().width / 2,
                                    gameResolution.y / 2 -
textMessage.getLocalBounds().height / 2);
        }

        if (state == State::GAME_OVER)
        {
            textMessage.setString("Press Enter to Start the Game !!");
            textMessage.setPosition(gameResolution.x / 2 -
textMessage.getLocalBounds().width / 2,
                                    gameResolution.y / 2 -
textMessage.getLocalBounds().height / 2);
        }

        // Draw the Scene
        window.clear();
        if (state == State::PAUSED)
```

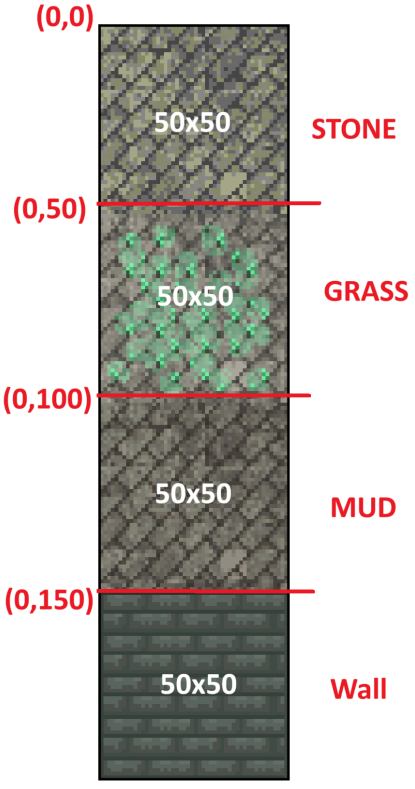
```
        {
            window.draw(textMessage);
        }
        if (state == State::PLAYING)
        {
            window.setView(gameView);

            window.setView(hudView);
            window.draw(textScore);
            window.draw(textHiScore);
        }
        if (state == State::LEVELING_UP)
        {
            window.draw(spriteBackground);
            window.draw(textMessage);
        }

        if (state == State::GAME_OVER)
        {
            window.draw(spriteBackground);
            window.draw(textMessage);
        }

        window.display();
    }

    return (0);
}
```

Working with Sprite Sheet for Designing Arena

Objective

- Design an Arena with given background Sprite Sheet.
- Types of backgrounds
  - grass
  - stone
  - mud
  - wall
- Wall will surround the entire arena and arena will be filled with random blocks of grass, stone, bush.

- A sprite sheet is a set of images, either frames of animation, or totally individual graphics contained in one image file.
- **Why it's better:**
  - **Reduces GPU calls:** Loading textures is expensive. Keeping one texture on the GPU is faster than switching many.
  - **Faster frame rendering**: No need to wait for texture loading between frames.
  - **Efficient memory usage:** Avoids repeated memory allocation and deallocation.
  - **Batch rendering:** Makes it easier to draw multiple sprites in one go, improving performance.
- Here, the following background sprite is used for 4 types of background namely grass, stone, mud and wall



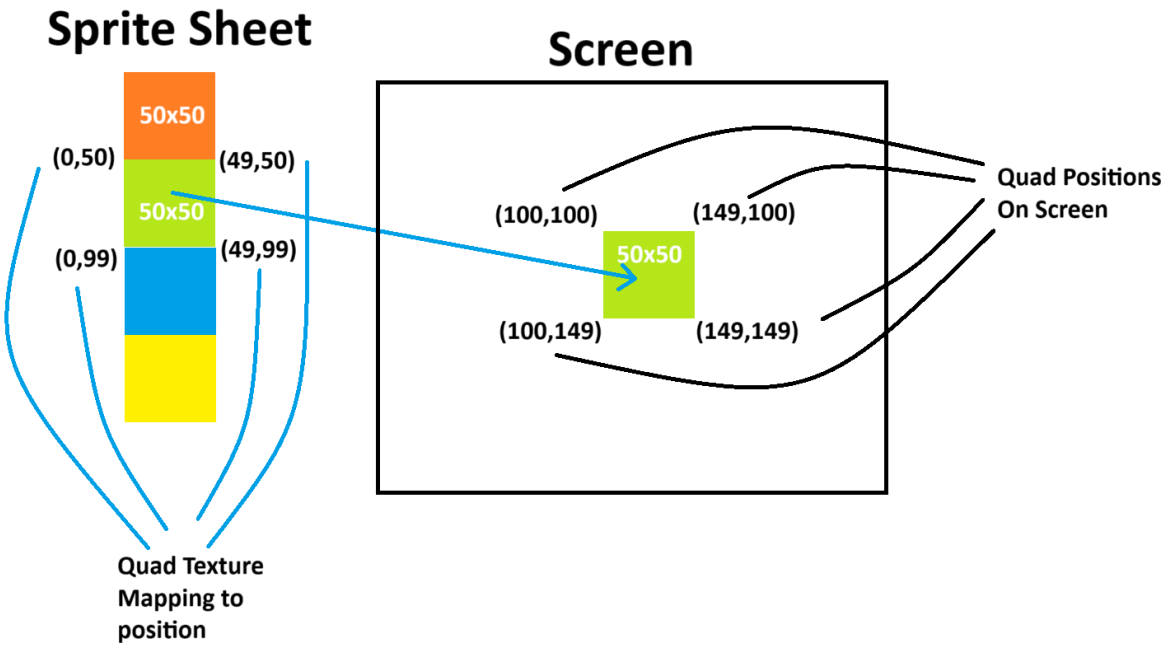
- Each tile in background can be a sprite or sprite array. However, it complicates setting and managing the tiles. The dynamicity of the arena size may also contribute to the complication. Hence **Vertex Array** is considered to avoid complication.
- `sf::VertexArray` : A dynamic array of vertices used to draw complex shapes (lines, quads, triangles, etc.) efficiently.
- Why Use Vertex Array?
  - High performance — all vertices are sent in a single draw call
  - Useful for tile maps, particle systems, custom shapes
  - Avoids multiple `sf::Sprite objects`
- Declaration Syntax

```
sf::VertexArray array(sf::Quads, 100); // 100 vertices forming 25 quads
```

- Vertex array can be created with primitive types like `sf::Quads`
  - Other primitive types available:

Type	Description
`sf::Points`	Just dots
`sf::Lines`	Unconnected lines (2 points each)
`sf::LineStrip`	Connected line path
`sf::Triangles`	Individual triangles (3 pts)
`sf::TriangleStrip`	Connected triangle mesh
`sf::TriangleFan`	Fan-like structure (circle)
`sf::Quads`	4-point rectangles

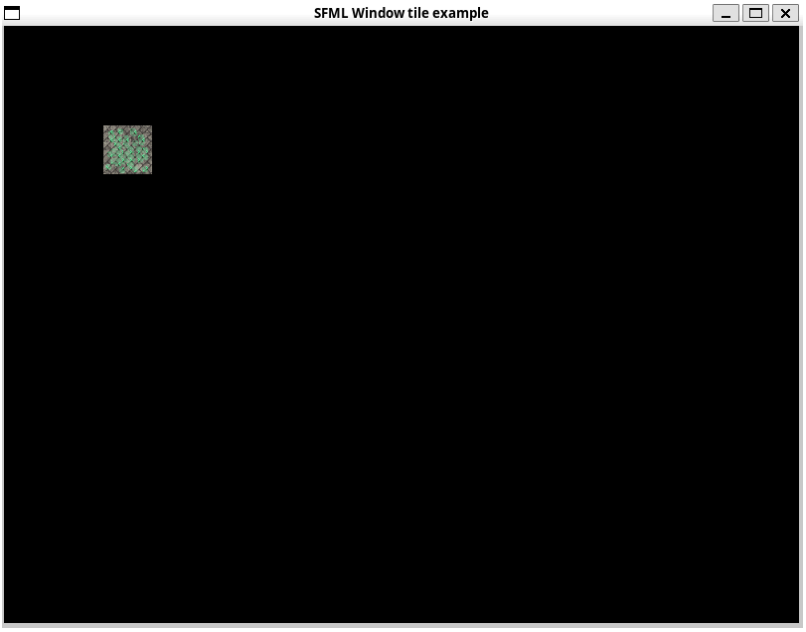
- `sf::Quads` represents each set of **4 consecutive vertices** forms a **rectangle** (quad). - It is used to apply a **portion of a texture** (e.g., one tile from a sprite sheet) to a quad by setting **texture coordinates**.
  Example




```
#include<SFML/Graphics.hpp>
using namespace sf;

int main()
{
    RenderWindow window(VideoMode(800, 600), "SFML Window tile example");

    Texture texture;
    texture.loadFromFile("background_sheet.png");

    VertexArray vt(Quads, 4);

    vt[0].position = Vector2f(100, 100);
    vt[1].position = Vector2f(149, 100);
    vt[2].position = Vector2f(149, 149);
    vt[3].position = Vector2f(100, 149);

    vt[0].texCoords = Vector2f(0, 50);
    vt[1].texCoords = Vector2f(49, 50);
    vt[2].texCoords = Vector2f(49, 99);
    vt[3].texCoords = Vector2f(0, 99);

    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event)) {
            if (event.type == Event::Closed) {
                window.close();
            }
        }
        window.clear();
```

```
        RenderStates states;
        states.texture = &texture;
        window.draw(vt, states);
        window.display();
    }
}
```

Output:



Creating a arena of 10x10 tiles

Requirements

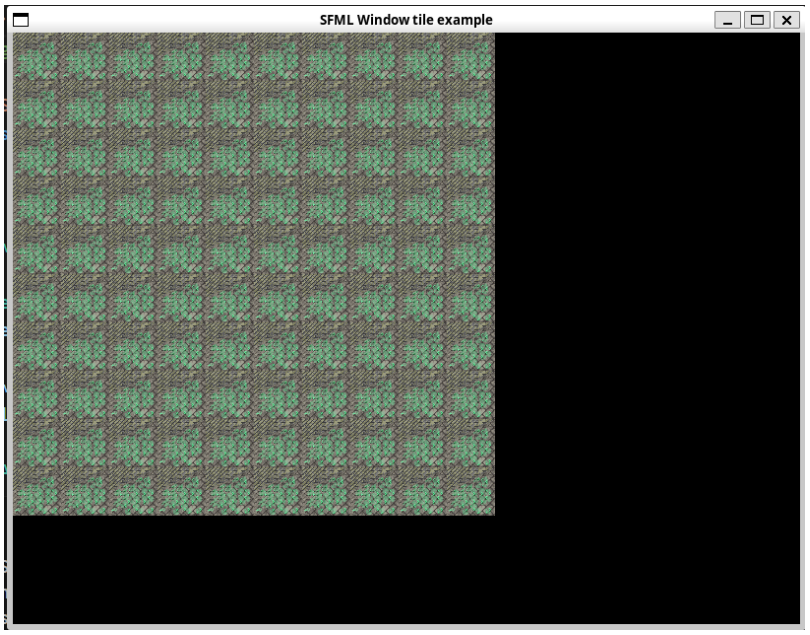
- For 1 tile 4 Quad data types are needed. (as seen in previous example)
- Hence, for 10x10 tiles we need 400 Quads.
- Vertex array will be used to create the 400 Quads.

Placing a single tile texture to entire 10x10 arena.

```
// Creating 10x10 tiles for arena

#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{
    RenderWindow window(VideoMode(800, 600), "SFML Window tile example");

    Texture texture;
    texture.loadFromFile("background_sheet.png");
```

```
    int rowTiles = 10;
    int columnTiles = 10;

    VertexArray vt(Quads, 4 * rowTiles * columnTiles);

    for (int row = 0; row < rowTiles; ++row)
    {
        for (int col = 0; col < columnTiles; ++col)
        {
            int index = (row * columnTiles + col) * 4; // start of tile
Quad
            vt[index + 0].position = Vector2f(col * 50, row * 50);
            vt[index + 1].position = Vector2f((col + 1) * 50-1, row *
50);
            vt[index + 2].position = Vector2f((col + 1) * 50-1, (row + 1)
* 50-1);
            vt[index + 3].position = Vector2f(col * 50, (row + 1) * 50-
1);

            vt[index + 0].texCoords = Vector2f(0, 0);
            vt[index + 1].texCoords = Vector2f(49, 50);
            vt[index + 2].texCoords = Vector2f(49, 99);
            vt[index + 3].texCoords = Vector2f(0, 99);
        }
    }

    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
            {
                window.close();
            }
        }
        window.clear();
        RenderStates states;
        states.texture = &texture;
        window.draw(vt, states);
        window.display();
    }
}
```

Output:

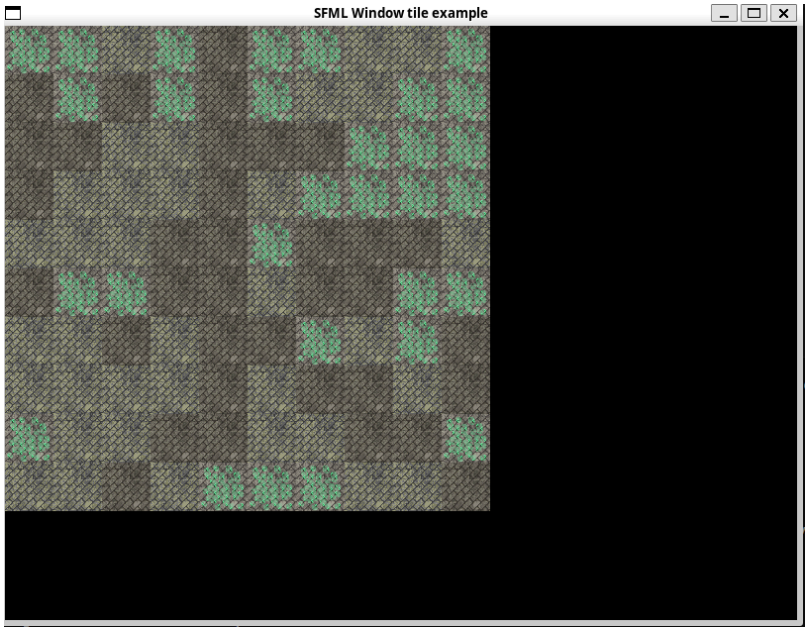


Placing random tile (stone, grass or mud) texture to entire 10x10 arena.

```
// Creating 10x10 tiles for arena

#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{
    RenderWindow window(VideoMode(800, 600), "SFML Window tile example");

    Texture texture;
    texture.loadFromFile("background_sheet.png");

    int rowTiles = 10;
    int columnTiles = 10;
    int titleType = 3; //0:Stone, 1: Grass, 2:Mud

    VertexArray vt(Quads, 4 * rowTiles * columnTiles);

    for (int row = 0; row < rowTiles; ++row)
    {
        for (int col = 0; col < columnTiles; ++col)
        {
            int index = (row * columnTiles + col) * 4;
            vt[index + 0].position = Vector2f(col * 50, row * 50);
            vt[index + 1].position = Vector2f((col + 1) * 50-1, row *
50);
            vt[index + 2].position = Vector2f((col + 1) * 50-1, (row + 1)
* 50-1);
            vt[index + 3].position = Vector2f(col * 50, (row + 1) * 50-
```

```
1);

            int randTileType = rand()%3; // getting value 0,1, or 2

            vt[index + 0].texCoords = Vector2f(0, randTileType*50);
            vt[index + 1].texCoords = Vector2f(49, randTileType*50);
            vt[index + 2].texCoords = Vector2f(49, (randTileType+1)*50-
1);
            vt[index + 3].texCoords = Vector2f(0, (randTileType+1)*50-1);
        }
    }

    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
            {
                window.close();
            }
        }
        window.clear();
        RenderStates states;
        states.texture = &texture;
        window.draw(vt, states);
        window.display();
    }
}
```

Output

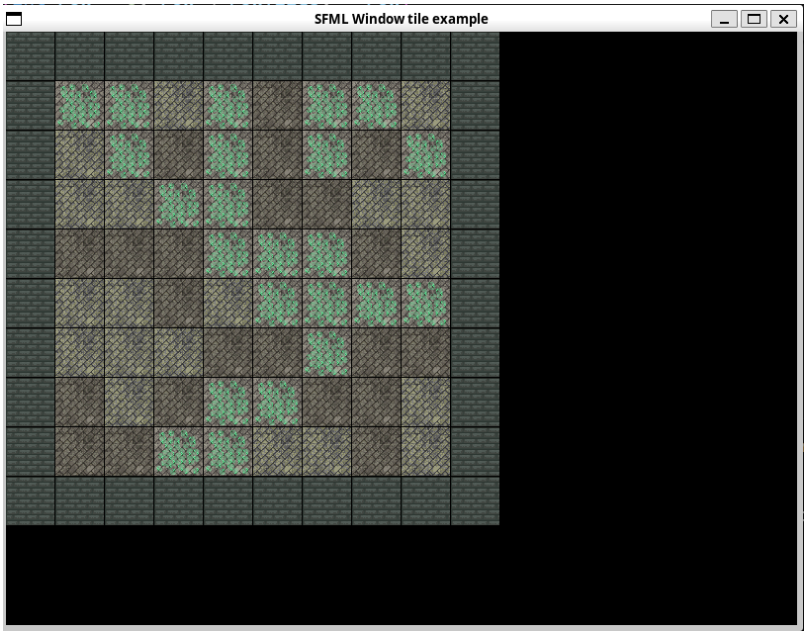


Making the Boundary with wall

```
// Creating 10x10 tiles for arena with boundary wall

#include <SFML/Graphics.hpp>
using namespace sf;

int main()
{
    RenderWindow window(VideoMode(800, 600), "SFML Window tile example");

    Texture texture;
    texture.loadFromFile("background_sheet.png");

    int rowTiles = 10;
    int columnTiles = 10;
    int titleType = 3; //0:Stone, 1: Grass, 2:Mud

    VertexArray vt(Quads, 4 * rowTiles * columnTiles);

    for (int row = 0; row < rowTiles; ++row)
    {
        for (int col = 0; col < columnTiles; ++col)
        {
            int index = (row * columnTiles + col) * 4;
            vt[index + 0].position = Vector2f(col * 50, row * 50);
            vt[index + 1].position = Vector2f((col + 1) * 50-1, row *
50);
            vt[index + 2].position = Vector2f((col + 1) * 50-1, (row + 1)
* 50-1);
            vt[index + 3].position = Vector2f(col * 50, (row + 1) * 50-
1);
            int TileType;

            if(row == 0 || col == 0 || row == rowTiles - 1 || col ==
columnTiles - 1 )
            {
                TileType = 3; // Using stone for the boundary
            }
            else{
                TileType = rand()%3; // getting value 0,1, or 2
            }

            vt[index + 0].texCoords = Vector2f(0, TileType*50);
            vt[index + 1].texCoords = Vector2f(49, TileType*50);
            vt[index + 2].texCoords = Vector2f(49, (TileType+1)*50-1);
            vt[index + 3].texCoords = Vector2f(0, (TileType+1)*50-1);
        }
    }

    while (window.isOpen())
```

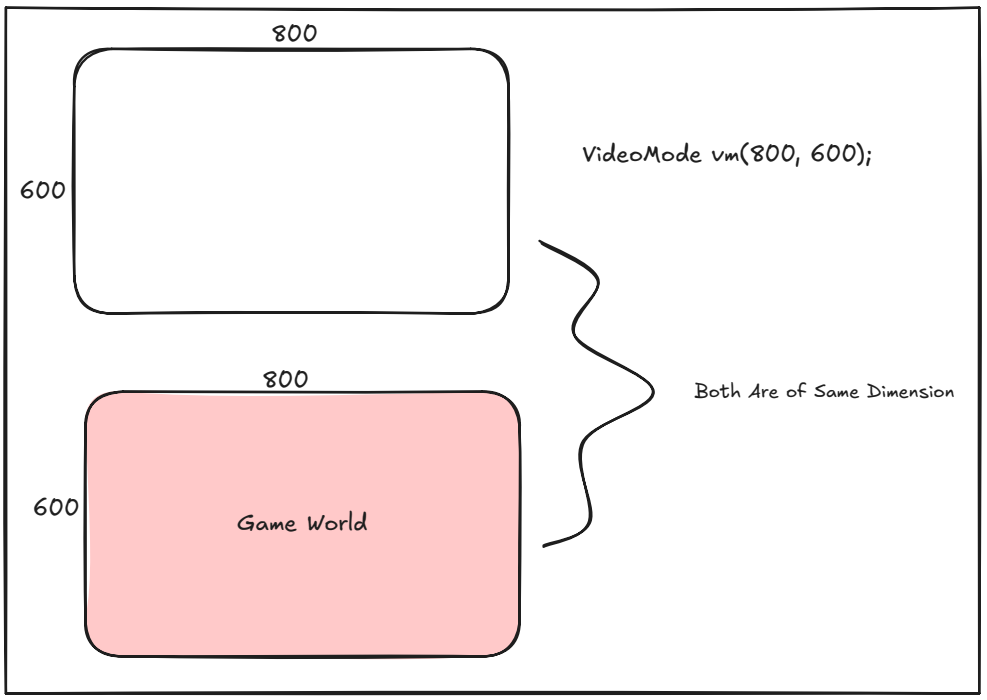
```
    {
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
            {
                window.close();
            }
        }
        window.clear();
        RenderStates states;
        states.texture = &texture;
        window.draw(vt, states);
        window.display();
    }
}
```

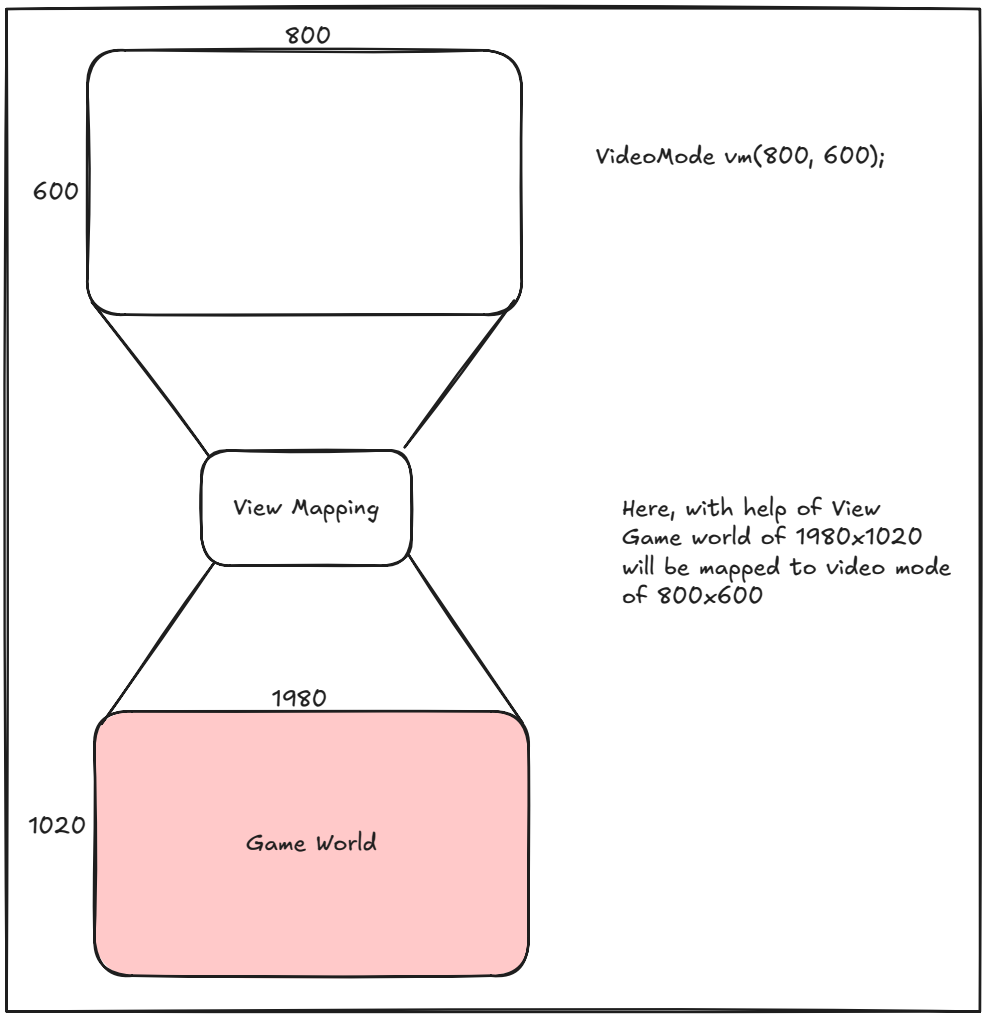
Output:



Working with View

- `sf::View` class plays a critical role in **controlling what part of the 2D world is visible on the screen**. It acts like a camera or window into the game world.
- Importance of sf::View in SFML
    1. Defines What to Show
        - Controls which section of your world is displayed in the render window.
        - Useful for large worlds or maps where only a portion should be visible.
    2. Enables Camera Movement
        - You can move the view (pan), zoom in/out, or rotate it to simulate a dynamic camera.
        - Great for player following, cutscenes, or map exploration.
    3. Decouples Game World from Screen Size
        - You can design your game world in world coordinates (e.g., 1000x1000) and still show only a portion that fits in the screen.
        - This makes it easier to handle different screen resolutions or window sizes.
    4. Allows UI Separation
        - You can set a fixed view for UI elements (like health bars or menus) that doesn't move with the game world.
        - Helps in organizing layers (game world vs. HUD).
    5. Supports Zooming and Scaling
        - view.zoom() lets you zoom into or out of the scene dynamically.
        - Useful for strategy games or maps.



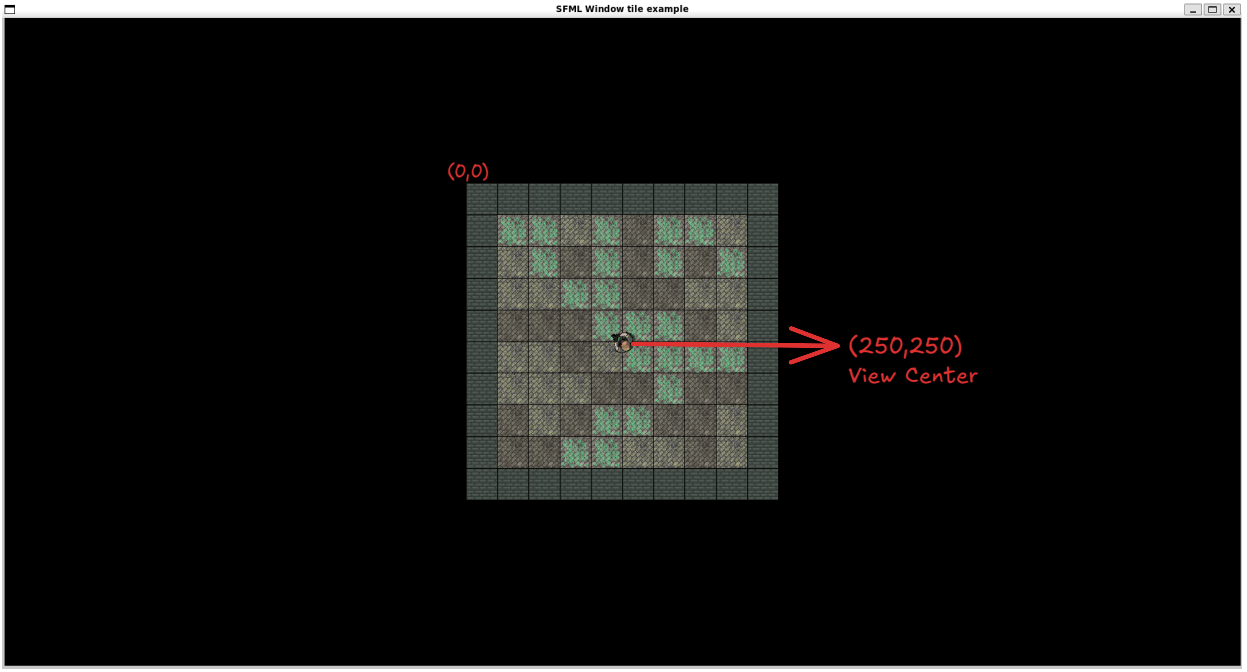



```
// Creating 10x10 tiles for arena with boundary wall

#include <SFML/Graphics.hpp>
#include <cmath>
using namespace sf;
```

```
int main()
{
    RenderWindow window(VideoMode(1980, 1020), "SFML Window tile
example");

    // Setup the view to be 500x500 and center it in the window
    View view(FloatRect(0, 0 ,1980, 1020));
    view.setCenter(500/2, 500/2); // Center on grid
    window.setView(view);


    Texture texture;
    texture.loadFromFile("background_sheet.png");

    int rowTiles = 10;
    int columnTiles = 10;
    int titleType = 3; //0:Stone, 1: Grass, 2:Mud
    float arenaWidth = rowTiles*50;
    float arenaHeight = columnTiles*50;

    VertexArray vt(Quads, 4 * rowTiles * columnTiles);

    for (int row = 0; row < rowTiles; ++row)
    {
        for (int col = 0; col < columnTiles; ++col)
        {
            int index = (row * columnTiles + col) * 4;
            vt[index + 0].position = Vector2f(col * 50, row * 50);
            vt[index + 1].position = Vector2f((col + 1) * 50-1, row *
50);
            vt[index + 2].position = Vector2f((col + 1) * 50-1, (row + 1)
* 50-1);
            vt[index + 3].position = Vector2f(col * 50, (row + 1) * 50-
1);
            int TileType;

            if(row == 0 || col == 0 || row == rowTiles - 1 || col ==
columnTiles - 1 )
            {
                TileType = 3; // Using stone for the boundary
            }
            else{
                TileType = rand()%3; // getting value 0,1, or 2
            }

            vt[index + 0].texCoords = Vector2f(0, TileType*50);
            vt[index + 1].texCoords = Vector2f(49, TileType*50);
            vt[index + 2].texCoords = Vector2f(49, (TileType+1)*50-1);
            vt[index + 3].texCoords = Vector2f(0, (TileType+1)*50-1);
        }
```

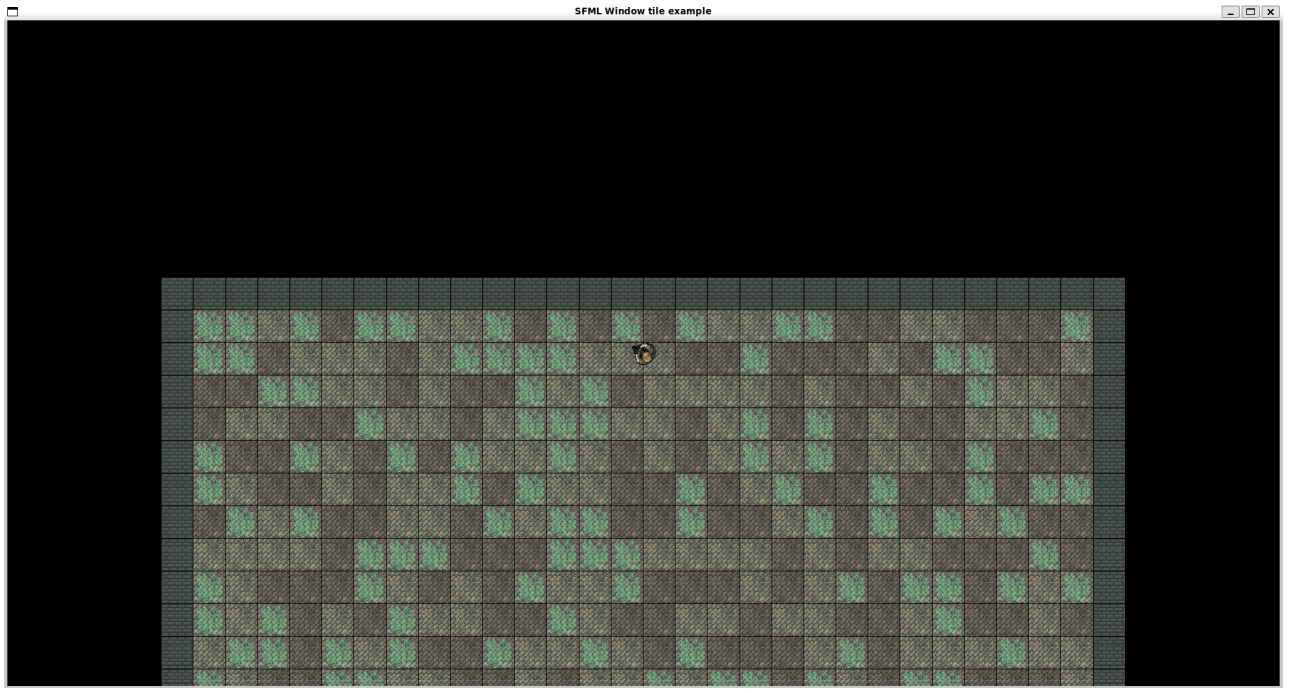
```
    }

    Sprite spritePlayer;
    Texture texturePlayer;
    texturePlayer.loadFromFile("player.png");
    spritePlayer.setTexture(texturePlayer);
    FloatRect playerBound = spritePlayer.getLocalBounds();
    spritePlayer.setOrigin(playerBound.width/2, playerBound.height/2);

    float playerSpeed = 5;
    float pixelPerSecPlayerX = window.getSize().x/  playerSpeed;
    float pixelPerSecPlayerY = window.getSize().y/  playerSpeed;

    Clock ct;
    Time dt;
    spritePlayer.setPosition(arenaWidth/2, arenaHeight/2);

    while (window.isOpen())
    {
        dt = ct.restart();
        Event ev;
        while(window.pollEvent(ev))
        {
            if(ev.type == Event::Closed)
            {
                window.close();
            }
        }
        float x = spritePlayer.getPosition().x;
        float y = spritePlayer.getPosition().y;
        if(Keyboard::isKeyPressed(Keyboard::A))
            x = x - dt.asSeconds() * pixelPerSecPlayerX;
        if(Keyboard::isKeyPressed(Keyboard::D))
            x = x + dt.asSeconds() * pixelPerSecPlayerX;
        if(Keyboard::isKeyPressed(Keyboard::W))
            y = y - dt.asSeconds() * pixelPerSecPlayerY;
        if(Keyboard::isKeyPressed(Keyboard::S))
            y = y + dt.asSeconds() * pixelPerSecPlayerY;
        if(x > 500)
            x = 500;
        if(y > 500)
            y = 500;
        if(x < 0)
            x = 0;
        if(y < 0)
            y = 0;
        spritePlayer.setPosition(x,y);

        Vector2i mousePos = Mouse::getPosition(window);
        Vector2f mouseWindowPos = window.mapPixelToCoords(mousePos);
```

```
        float dx = mouseWindowPos.x - x;
        float dy =mouseWindowPos.y - y;

        float angle = atan2(dy, dx) * (180.0/3.141159f);
        spritePlayer.setRotation(angle);

        window.clear();
        RenderStates states;
        states.texture = &texture;
        window.draw(vt, states);
        window.draw(spritePlayer);
        window.display();
    }
}
```

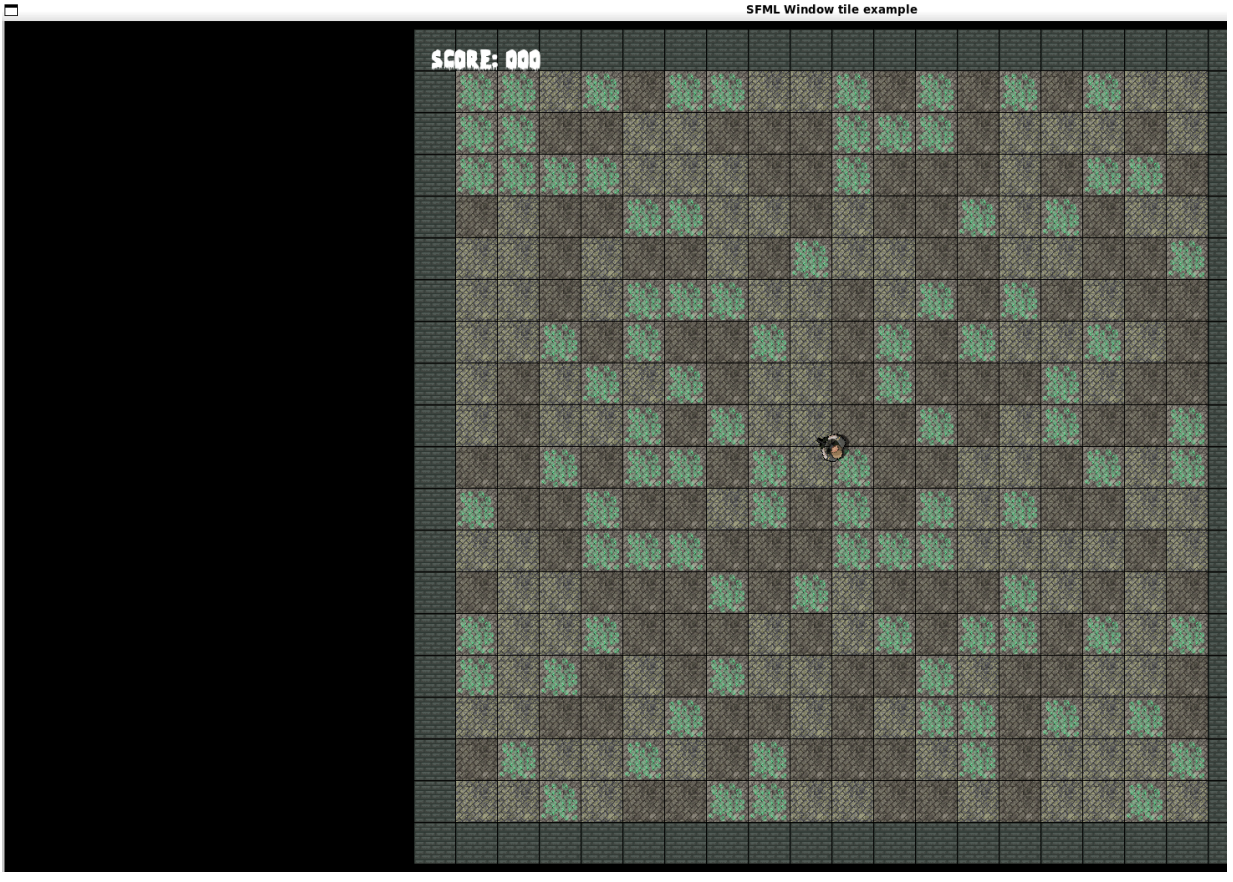


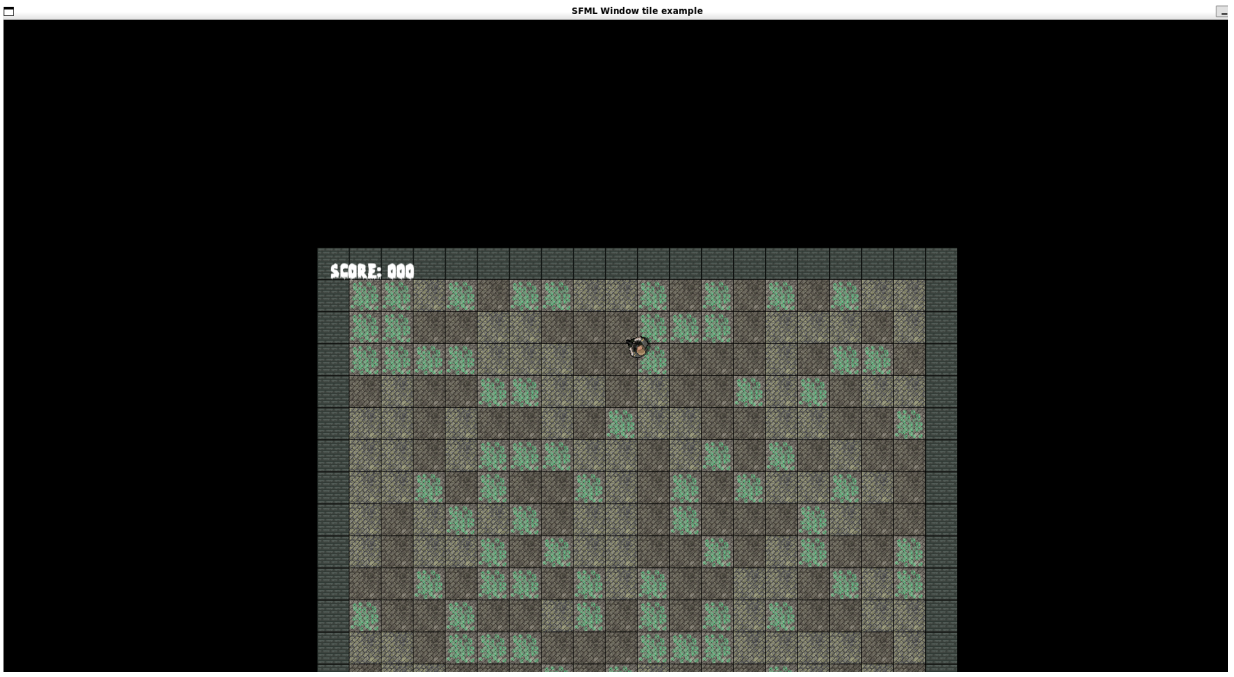
- Sometime the game world can be bigger than the visible frame. In such a case, we want to move the arena with the player movement. We must set the center of the view to be the player center.

```
view.setCenter(spritePlayer.getPosition());
window.setView(view);
```



- The HUD is drawn in same view HUD will also move with the player movement. For example :





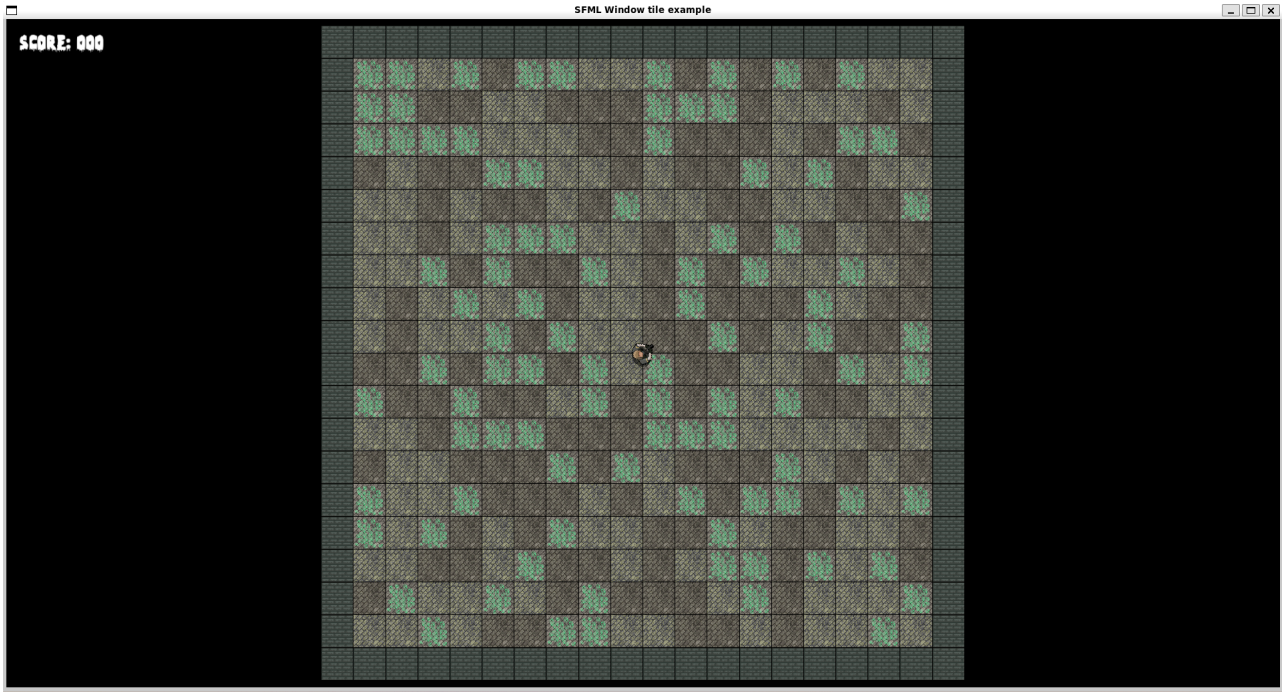
- We can see the Score: 000 HUD moved with the player. We want to set score HUD to be fixed at the top. Hence, we need tow separate views
  1. The player View which will move with the player.
  2. The HUD view which will remain fixed to display the HUD at a fixed position

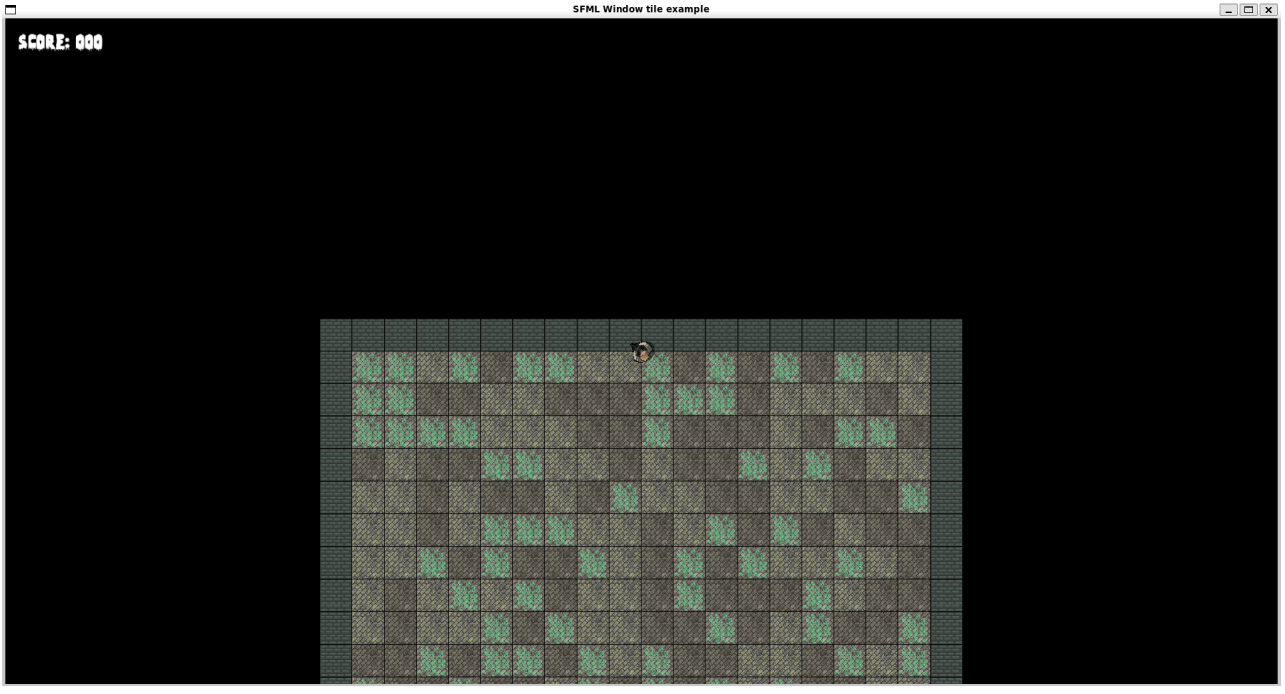
```
// Creating 10x10 tiles for arena with boundary wall

#include <SFML/Graphics.hpp>
#include <cmath>
using namespace sf;

int main() {
  RenderWindow window(VideoMode(800, 600), "SFML Window tile example");

  // Setup the view to be 500x500 and center it in the window
  View view(FloatRect(0, 0, 1980, 1020));
  View hudView(FloatRect(0, 0, 1980, 1020)); // stays static, doesn't
move with the player

  Vector2f arenaSize(1000, 1000);

  view.setCenter(arenaSize.x / 2, arenaSize.y / 2); // Center on grid
  window.setView(view);

  Texture texture;
  texture.loadFromFile("background_sheet.png");

  int rowTiles = arenaSize.x / 50;
  int columnTiles = arenaSize.y / 50;
  int titleType = 3; // 0:Stone, 1: Grass, 2:Mud
```

```
  int scoreValue = 0;
  int health = 0;



  // Example HUD element: simple score text
  Text scoreText;
  Font font;
  font.loadFromFile("zombiecontrol.ttf"); // Make sure you have the font
file
  scoreText.setFont(font);
  scoreText.setString("Score: 000");
  scoreText.setCharacterSize(30);
  scoreText.setFillColor(Color::White);
  scoreText.setPosition(20, 20); // top-left corner


  VertexArray vt(Quads, 4 * rowTiles * columnTiles);

  for (int row = 0; row < rowTiles; ++row) {
    for (int col = 0; col < columnTiles; ++col) {
      int index = (row * columnTiles + col) * 4;
      vt[index + 0].position = Vector2f(col * 50, row * 50);
      vt[index + 1].position = Vector2f((col + 1) * 50 - 1, row * 50);
      vt[index + 2].position = Vector2f((col + 1) * 50 - 1, (row + 1) *
50 - 1);
      vt[index + 3].position = Vector2f(col * 50, (row + 1) * 50 - 1);
      int TileType;

      if (row == 0 || col == 0 || row == rowTiles - 1 ||
          col == columnTiles - 1) {
        TileType = 3; // Using stone for the boundary
      } else {
        TileType = rand() % 3; // getting value 0,1, or 2
      }

      vt[index + 0].texCoords = Vector2f(0, TileType * 50);
      vt[index + 1].texCoords = Vector2f(49, TileType * 50);
      vt[index + 2].texCoords = Vector2f(49, (TileType + 1) * 50 - 1);
      vt[index + 3].texCoords = Vector2f(0, (TileType + 1) * 50 - 1);
    }
  }

  Sprite spritePlayer;
  Texture texturePlayer;
  texturePlayer.loadFromFile("player.png");
  spritePlayer.setTexture(texturePlayer);
  FloatRect playerBound = spritePlayer.getLocalBounds();
  spritePlayer.setOrigin(playerBound.width / 2, playerBound.height / 2);
```

```
float playerSpeed = 5;
float pixelPerSecPlayerX = window.getSize().x / playerSpeed;
float pixelPerSecPlayerY = window.getSize().y / playerSpeed;

Clock ct;
Time dt;
spritePlayer.setPosition(arenaSize.x / 2, arenaSize.y / 2);

while (window.isOpen()) {
  dt = ct.restart();
  Event ev;
  while (window.pollEvent(ev)) {
    if (ev.type == Event::Closed) {
      window.close();
    }
  }
  float x = spritePlayer.getPosition().x;
  float y = spritePlayer.getPosition().y;
  if (Keyboard::isKeyPressed(Keyboard::A))
    x = x - dt.asSeconds() * pixelPerSecPlayerX;
  if (Keyboard::isKeyPressed(Keyboard::D))
    x = x + dt.asSeconds() * pixelPerSecPlayerX;
  if (Keyboard::isKeyPressed(Keyboard::W))
    y = y - dt.asSeconds() * pixelPerSecPlayerY;
  if (Keyboard::isKeyPressed(Keyboard::S))
    y = y + dt.asSeconds() * pixelPerSecPlayerY;
  if (x > arenaSize.x - 50)
    x = arenaSize.x - 50;
  if (y > arenaSize.y - 50)
    y = arenaSize.y - 50;
  if (x < 50)
    x = 50;
  if (y < 50)
    y = 50;
  spritePlayer.setPosition(x, y);

  view.setCenter(spritePlayer.getPosition());
  window.setView(view);

  Vector2i mousePos = Mouse::getPosition(window);
  Vector2f mouseWindowPos = window.mapPixelToCoords(mousePos);

  float dx = mouseWindowPos.x - x;
  float dy = mouseWindowPos.y - y;

  float angle = atan2(dy, dx) * (180.0 / 3.141159f);
  spritePlayer.setRotation(angle);

  window.clear();
  RenderStates states;
```

```
    states.texture = &texture;
    window.draw(vt, states);
    window.draw(spritePlayer);

    window.setView(hudView); // Adding The score text using hudView.
    window.draw(scoreText);
    window.display();
```





Building Player Class

```
//Player Class
#pragma once
#include <cmath>
#include <SFML/Graphics.hpp>
using namespace sf;
class Player
{
private:
    const float START_SPEED = 200;
    const float START_HEALTH = 100;

    // Where is the player
    Vector2f m_Position;

    // Of course we will need a sprite
    Sprite m_Sprite;

    // And a texture
    // !!Watch this space!!
    Texture m_Texture;

    // What is the screen resolution
    Vector2f m_Resolution;

    // What size is the current arena
    IntRect m_Arena;

    // How big is each tile of the arena
    int m_TileSize;

    // Which directions is the player currently moving in
    bool m_UpPressed;
    bool m_DownPressed;
    bool m_LeftPressed;
    bool m_RightPressed;

    // How much health has the player got?
    int m_Health;
    // What is the maximum health the player can have
    int m_MaxHealth;

    // When was the player last hit
    Time m_LastHit;

    // Speed in pixels per second
    float m_Speed;
```

```
    // All our public functions will come next
public:
    Player();

    void spawn(IntRect arena, Vector2f resolution, int tileSize);

    // Handle the player getting hit by a zombie
    bool hit(Time timeHit);

    // How long ago was the player last hit
    Time getLastHitTime();

    // Where is the player
    FloatRect getPosition();

    // Where is the center of the player
    Vector2f getCenter();

    // Which angle is the player facing
    float getRotation();

    // Send a copy of the sprite to main
    Sprite getSprite();

    // How much health has the player currently got?
    int getHealth();

    // The next four functions move the player
    void moveLeft();

    void moveRight();

    void moveUp();

    void moveDown();

    // Stop the player moving in a specific direction
    void stopLeft();

    void stopRight();

    void stopUp();

    void stopDown();

    // We will call this function once every frame
    void update(float elapsedTime, Vector2i mousePosition);

    // Give player a speed boost
```

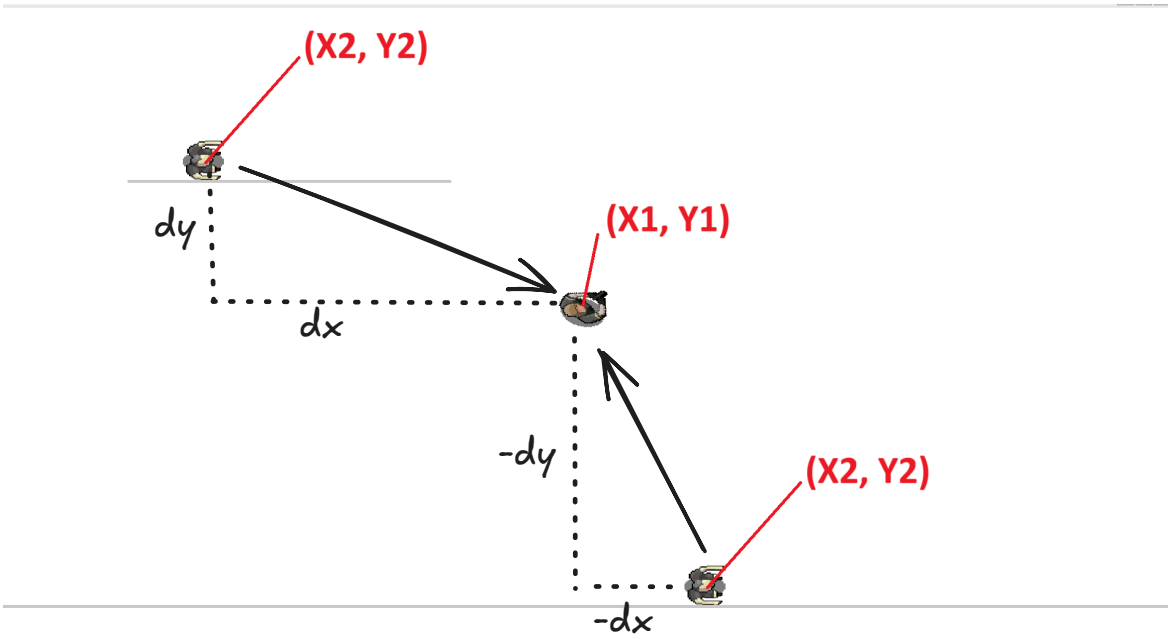
```
    void upgradeSpeed();

    // Give the player some health
    void upgradeHealth();

    // Increase the maximum amount of health the player can have
    void increaseHealthLevel(int amount);
    void resetPlayerStats();
};

Player::Player()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;

    // Associate a texture with the sprite
    // !!Watch this space!!
    m_Texture.loadFromFile("player.png");
    m_Sprite.setTexture(m_Texture);

    // Set the origin of the sprite to the centre,
    // for smooth rotation
    m_Sprite.setOrigin(25, 25);
    // m_Sprite.setPosition(1920/2.0f, 1080/2.0f);
}

void Player::spawn(IntRect arena, Vector2f resolution, int tileSize)
{
    // Place the player in the middle of the arena
    m_Position.x = arena.width / 2;
    m_Position.y = arena.height / 2;

    // Copy the details of the arena to the player's m_Arena
    m_Arena.left = arena.left;
    m_Arena.width = arena.width;
    m_Arena.top = arena.top;
    m_Arena.height = arena.height;

    // Remember how big the tiles are in this arena
    m_TileSize = tileSize;

    // Strore the resolution for future use
    m_Resolution.x = resolution.x;
    m_Resolution.y = resolution.y;
}

Time Player::getLastHitTime()
{
    return m_LastHit;
```

```
}

bool Player::hit(Time timeHit)
{
    if (timeHit.asMilliseconds() - m_LastHit.asMilliseconds() > 200) // 2
tenths of second
    {
        m_LastHit = timeHit;
        m_Health -= 1;
        return true;
    }
    else
    {
        return false;
    }
}

FloatRect Player::getPosition()
{
    return m_Sprite.getGlobalBounds();
}

Vector2f Player::getCenter()
{
    return m_Position;
}

float Player::getRotation()
{
    return m_Sprite.getRotation();
}

Sprite Player::getSprite()
{
    return m_Sprite;
}

int Player::getHealth()
{
    return m_Health;
}

void Player::moveLeft()
{
    m_LeftPressed = true;
}

void Player::moveRight()
{
    m_RightPressed = true;
```

```
}

void Player::moveUp()
{
    m_UpPressed = true;
}

void Player::moveDown()
{
    m_DownPressed = true;
}

void Player::stopLeft()
{
    m_LeftPressed = false;
}

void Player::stopRight()
{
    m_RightPressed = false;
}

void Player::stopUp()
{
    m_UpPressed = false;
}

void Player::stopDown()
{
    m_DownPressed = false;
}

void Player::update(float elapsedTime, Vector2i mousePosition)
{

    if (m_UpPressed)
    {
        m_Position.y -= m_Speed * elapsedTime;
    }

    if (m_DownPressed)
    {
        m_Position.y += m_Speed * elapsedTime;
    }

    if (m_RightPressed)
    {
        m_Position.x += m_Speed * elapsedTime;
    }
```

```
    if (m_LeftPressed)
    {
        m_Position.x -= m_Speed * elapsedTime;
    }

    m_Sprite.setPosition(m_Position);

    // Keep the player in the arena
    if (m_Position.x > m_Arena.width - m_TileSize)
    {
        m_Position.x = m_Arena.width - m_TileSize;
    }

    if (m_Position.x < m_Arena.left + m_TileSize)
    {
        m_Position.x = m_Arena.left + m_TileSize;
    }

    if (m_Position.y > m_Arena.height - m_TileSize)
    {
        m_Position.y = m_Arena.height - m_TileSize;
    }

    if (m_Position.y < m_Arena.top + m_TileSize)
    {
        m_Position.y = m_Arena.top + m_TileSize;
    }

    // Calculate the angle the player is facing
    float angle = (atan2(mousePosition.y - m_Resolution.y / 2,
                         mousePosition.x - m_Resolution.x / 2) *
                   180) /
                  3.141;

    m_Sprite.setRotation(angle);
}

void Player::upgradeSpeed()
{
    // 20% speed upgrade
    m_Speed += (START_SPEED * .2);
}

void Player::upgradeHealth()
{
    // 20% max health upgrade
    m_MaxHealth += (START_HEALTH * .2);
}

void Player::increaseHealthLevel(int amount)
```

```
{
    m_Health += amount;

    // But not beyond the maximum
    if (m_Health > m_MaxHealth)
    {
        m_Health = m_MaxHealth;
    }
}
void Player::resetPlayerStats()
{
    m_Speed = START_SPEED;
    m_Health = START_HEALTH;
    m_MaxHealth = START_HEALTH;
}
```




```
#include<SFML/Graphics.hpp>
#include<iostream>
#include<cmath>
using namespace sf;
using namespace std;

int main()
{
    VideoMode video(800, 600);
    RenderWindow window(video, "Zombie Arena !!!");

    Sprite spritePlayer;
    Texture texturePlayer;
    texturePlayer.loadFromFile("player.png");
    spritePlayer.setTexture(texturePlayer);
    FloatRect playerBound = spritePlayer.getLocalBounds();
    spritePlayer.setOrigin(playerBound.width/2, playerBound.height/2);

    float playerSpeed = 5;
    float pixelPerSecPlayerX = window.getSize().x/  playerSpeed;
    float pixelPerSecPlayerY = window.getSize().y/  playerSpeed;

    Sprite spriteZombie;
    Texture textureZombie;
    textureZombie.loadFromFile("chaser.png");
    spriteZombie.setTexture(textureZombie);
    FloatRect zombieBound = spriteZombie.getLocalBounds();
    spriteZombie.setOrigin(zombieBound.width/2, zombieBound.height/2);

    float zombieSpeed = 7;
```

```
    float pixelPerSecZombieX = window.getSize().x/  zombieSpeed;
    float pixelPerSecZombieY = window.getSize().y/  zombieSpeed;

    Clock ct;
    Time dt;

    spritePlayer.setPosition(window.getSize().x/2, window.getSize().y/2);
    spriteZombie.setPosition(rand()%window.getSize().x,
rand()%window.getSize().y);

    while(window.isOpen())
    {
        dt = ct.restart();
        Event ev;
        while(window.pollEvent(ev)){
            if(ev.type == Event::Closed)
                window.close();
        }

        //Move Zombite to player logic
        Vector2f playerXY = spritePlayer.getPosition();
        Vector2f zombieXY = spriteZombie.getPosition();
        if(playerXY.x > zombieXY.x)
            zombieXY.x = zombieXY.x + dt.asSeconds() *
pixelPerSecZombieX;
        else if (playerXY.x < zombieXY.x)
            zombieXY.x = zombieXY.x - dt.asSeconds() *
pixelPerSecZombieX;

        if(playerXY.y > zombieXY.y)
            zombieXY.y = zombieXY.y + dt.asSeconds() *
pixelPerSecZombieY;
        else if (playerXY.y < zombieXY.y)
            zombieXY.y = zombieXY.y - dt.asSeconds() *
pixelPerSecZombieY;

        spriteZombie.setPosition(zombieXY);

        dx = playerXY.x - zombieXY.x;
        dy = playerXY.y - zombieXY.y;

        angle = atan2(dy, dx) * (180.0/3.141159f);
        spriteZombie.setRotation(angle);

        window.clear(Color::White);
        window.draw(spritePlayer);
        window.draw(spriteZombie);
        window.display();
    }
```

```
    return(0);
}
```

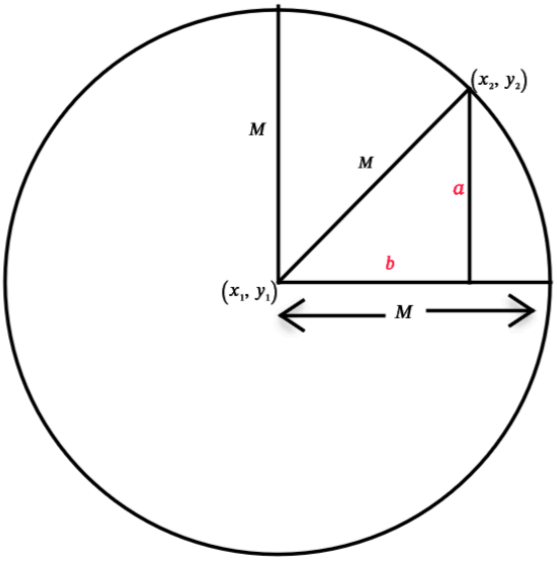
Zombie Class

```
#pragma once
#include <SFML/Graphics.hpp>
#include <cmath>

using namespace sf;

class Zombie
{
private:
    const float BLOATER_SPEED = 20;
    const float CHASER_SPEED = 40;
    const float CRAWLER_SPEED = 10;

    const float BLOATER_HEALTH = 5;
    const float CHASER_HEALTH = 1;
    const float CRAWLER_HEALTH = 3;

    Vector2f m_Position;
    Sprite m_Sprite;
    Texture m_Texture;
    float m_Speed;
    float m_Health;
    bool m_Alive = false;

public:
    FloatRect getPosition();
    Sprite getSprite();
    bool isAlive();
    void spawn(float startX, float startY, int type, int seed);
    void update(float elapsedTime, Vector2f playerLocation);
    bool hit();
};
FloatRect Zombie::getPosition()
{
    return m_Sprite.getGlobalBounds();
}
Sprite Zombie::getSprite()
{
    return m_Sprite;
}
bool Zombie::isAlive()
{
    return m_Alive;
```

```
}
void Zombie::spawn(float startX, float startY, int type, int seed)
{
    m_Position.x = startX;
    m_Position.y = startY;
    // m_Sprite.setPosition(m_Position);
    m_Sprite.setOrigin(25, 25);
    switch (type)
    {
    case 0:
        m_Texture.loadFromFile("bloater.png");
        m_Sprite.setTexture(m_Texture);
        m_Speed = BLOATER_SPEED;
        m_Health = BLOATER_HEALTH;
        m_Alive = true;
        break;

    case 1:
        m_Texture.loadFromFile("chaser.png");
        m_Sprite.setTexture(m_Texture);
        m_Speed = CHASER_SPEED;
        m_Health = CHASER_HEALTH;
        m_Alive = true;
        break;

    case 2:
        m_Texture.loadFromFile("crawler.png");
        m_Sprite.setTexture(m_Texture);
        m_Speed = CRAWLER_SPEED;
        m_Health = CRAWLER_HEALTH;
        m_Alive = true;
        break;
    }
    srand((int)time(0) * seed);
    float modifier = (rand() % (101 - 70) + 70);
    modifier = modifier / 100;
    m_Speed = m_Speed * modifier;
}
void Zombie::update(float elapsedTime, Vector2f playerLocation)
{
    if (m_Alive)
    {
        float playerX = playerLocation.x;
        float playerY = playerLocation.y;
        if (m_Position.x < playerX)
        {
            m_Position.x = m_Position.x + m_Speed * elapsedTime;
        }
        if (m_Position.x > playerX)
        {
```

```
            m_Position.x = m_Position.x - m_Speed * elapsedTime;
        }
        if (m_Position.y < playerY)
        {
            m_Position.y = m_Position.y + m_Speed * elapsedTime;
        }
        if (m_Position.y > playerY)
        {
            m_Position.y = m_Position.y - m_Speed * elapsedTime;
        }
        m_Sprite.setPosition(m_Position);
        float angle = (atan2(playerY - m_Position.y, playerX -
m_Position.x) * 180) / 3.141;
        m_Sprite.setRotation(angle);
    }
}
bool Zombie::hit()
{
    m_Health--;
    if (m_Health < 0)
    {
        m_Alive = false;
        m_Texture.loadFromFile("blood.png");
        m_Sprite.setTexture(m_Texture);
        return true;
    }
    return false;
}
```

Bullet Movement Logic



M is pixels covered by bullet in unit time.

$b = x_2 - x_1$

$a = y_2 - y_1$

$g = \frac{x_2 - x_1}{y_2 - y_1}$

For a:

$a^2 + b^2 = M^2$

$\Rightarrow (y_2 - y_1)^2 + (x_2 - x_1)^2 = M^2$

$\Rightarrow \frac{(y_2 - y_1)^2 + (x_2 - x_1)^2}{(y_2 - y_1)^2} = \frac{M^2}{(y_2 - y_1)^2}$

$\Rightarrow 1 + \frac{(x_2 - x_1)^2}{(y_2 - y_1)^2} = \frac{M^2}{(y_2 - y_1)^2}$

$\Rightarrow 1 + g^2 = \frac{M^2}{(y_2 - y_1)^2}$

$\Rightarrow (y_2 - y_1)^2 = \frac{M^2}{1 + g^2}$

$\Rightarrow y_2 - y_1 = \frac{M}{\sqrt{1 + g^2}}$

$\Rightarrow a = \frac{M}{\sqrt{1 + g^2}} \approx \frac{M}{1 + g}$

Similarly,

$a^2 + b^2 = M^2$

$\Rightarrow (y_2 - y_1)^2 + (x_2 - x_1)^2 = M^2$

$\Rightarrow \frac{(y_2 - y_1)^2 + (x_2 - x_1)^2}{(x_2 - x_1)^2} = \frac{M^2}{(x_2 - x_1)^2}$

$\Rightarrow \frac{(y_2 - y_1)^2}{(x_2 - x_1)^2} + 1 = \frac{M^2}{(x_2 - x_1)^2}$

$$\Rightarrow \frac{1}{g^2} + 1 = \frac{M^2}{(x_2 - x_1)^2}$$
$$\Rightarrow \frac{1 + g^2}{g^2} = \frac{M^2}{(x_2 - x_1)^2}$$
$$\Rightarrow (x_2 - x_1)^2 = \frac{M^2 g^2}{1 + g^2}$$
$$\Rightarrow x_2 - x_1 = \frac{M}{\sqrt{1 + g^2}} g$$
$$\Rightarrow b = ag$$

Bullet Class

```
#pragma once
#include <SFML/Graphics.hpp>
#include <cmath>
using namespace sf;

class Bullet
{
private:
    Vector2f m_Position;
    RectangleShape m_BulletShape;
    bool m_InFlight = false;
    float m_BulletSpeed = 1000;
    float m_BulletDistanceX;
    float m_BulletDistanceY;
    float m_XTarget;
    float m_YTarget;
    float m_MaxX;
    float m_MinX;
    float m_MaxY;
    float m_MinY;

public:
    Bullet();
    void stop();
    bool isInFlight();
    void shoot(float startX, float startY, float xTarget, float yTarget);
    FloatRect getPosition();
    RectangleShape getShape();
    void update(float elapsedTime);
};
Bullet::Bullet()
{
    m_BulletShape.setSize(sf::Vector2f(2, 2));
}
void Bullet::shoot(float startX, float startY, float targetX, float
targetY)
```

```
{
    m_InFlight = true;
    m_Position.x = startX;
    m_Position.y = startY;
    float gradient = (startX - targetX) / (startY - targetY);

    if (gradient < 0)
    {
        gradient *= -1;
    }
    m_BulletDistanceY = m_BulletSpeed / (1 + gradient);
    m_BulletDistanceX = m_BulletSpeed * (gradient / (1 + gradient));
    // float ratioXY = m_BulletSpeed / (1 + gradient);

    //  m_BulletDistanceY = m_BulletSpeed*(1/1+;m_BulletDistanceX =
ratioXY * gradient;
    if (targetX < startX)
    {
        m_BulletDistanceX *= -1;
    }

    if (targetY < startY)
    {
        m_BulletDistanceY *= -1;
    }

    /*float delX = targetX - startX;
    float delY = targetY - startY;
    float dist = sqrt(delX*delX + delY*delY);
    m_BulletDistanceY = m_BulletSpeed*delY/dist;
    m_BulletDistanceX = m_BulletSpeed*delX/dist;
    */

    // Finally, assign the results to the
    // member variables
    // m_XTarget = targetX;
    // m_YTarget = targetY;

    // Set a max range of 1000 pixels
    float range = 1000;
    m_MinX = startX - range;
    m_MaxX = startX + range;
    m_MinY = startY - range;
    m_MaxY = startY + range;

    // Position the bullet ready to be drawn
    m_BulletShape.setPosition(m_Position);
}
void Bullet::stop()
{
```

```
    m_InFlight = false;
}

bool Bullet::isInFlight()
{
    return m_InFlight;
}

FloatRect Bullet::getPosition()
{
    return m_BulletShape.getGlobalBounds();
}

RectangleShape Bullet::getShape()
{
    return m_BulletShape;
}

void Bullet::update(float elapsedTime)
{
    // Update the bullet position variables
    m_Position.x += m_BulletDistanceX * elapsedTime;
    m_Position.y += m_BulletDistanceY * elapsedTime;

    // Move the bullet
    m_BulletShape.setPosition(m_Position);

    // Has the bullet gone out of range?
    if (m_Position.x < m_MinX || m_Position.x > m_MaxX ||
        m_Position.y < m_MinY || m_Position.y > m_MaxY)
    {
        m_InFlight = false;
    }
}
```

PickUp Class

```
#pragma once
#include <SFML/Graphics.hpp>
using namespace sf;
class Pickup
{
private:
    const int HEALTH_START_VALUE = 50;
    const int AMMO_START_VALUE = 12;
    const int START_WAIT_TIME = 10;
    const int START_SECONDS_TO_LIVE = 5;
    Vector2f m_Position;
    Texture m_Texture;
    Sprite m_Sprite;
    int m_Type; // 1= health, 2-ammo
    int m_Value;
    IntRect m_Arena;
    bool m_Spawned;
    float m_SecondsSinceSpawn;
    float m_SecondsSinceDeSpawn;
    float m_SecondsToLive;
    float m_SecondsToWait;

public:
    Pickup(int type);
    FloatRect getPosition();
    Sprite getSprite();
    void spawn();
    bool isSpawned();
    void update(float elapsedTime);
    void setArena(IntRect arena);
    void upgrade();
    int gotIt();
};
Pickup::Pickup(int type)
{
    m_Type = type;
    if (m_Type == 1)
    { // HEALTH
        m_Texture.loadFromFile("health_pickup.png");
        m_Sprite.setTexture(m_Texture);
        m_Value = HEALTH_START_VALUE;
    }
    else
    {
        m_Texture.loadFromFile("ammo_pickup.png");
        m_Sprite.setTexture(m_Texture);
```

```
        m_Value = AMMO_START_VALUE;
    }
    m_Sprite.setOrigin(25, 25);
    m_SecondsToLive = START_SECONDS_TO_LIVE;
    m_SecondsToWait = START_WAIT_TIME;
}
void Pickup::setArena(IntRect arena)
{
    m_Arena.left = arena.left + 50;
    m_Arena.top = arena.top + 50;
    m_Arena.width = arena.width - 50;
    m_Arena.height = arena.height - 50;
    spawn();
}
FloatRect Pickup::getPosition()
{
    return m_Sprite.getGlobalBounds();
}
Sprite Pickup::getSprite()
{
    return m_Sprite;
}
bool Pickup::isSpawned()
{
    return m_Spawned;
}
void Pickup::spawn()
{
    srand((int)time(0) / m_Type);
    m_Position.x = rand() % m_Arena.width;
    srand((int)time(0) * m_Type);
    m_Position.y = rand() % m_Arena.height;
    m_Sprite.setPosition(m_Position);
    m_SecondsSinceSpawn = 0;
    m_Spawned = true;
}
void Pickup::update(float elapsedTime)
{
    if (m_Spawned)
    {
        m_SecondsSinceSpawn = m_SecondsSinceSpawn + elapsedTime;
    }
    else
    {
        m_SecondsSinceDeSpawn = m_SecondsSinceDeSpawn + elapsedTime;
    }
    if (m_SecondsSinceDeSpawn > m_SecondsToWait && !m_Spawned)
    {
        spawn();
    }
```

```
    if (m_SecondsSinceSpawn > m_SecondsToLive && m_Spawned)
    {
        m_Spawned = false;
        m_SecondsSinceDeSpawn = 0;
    }
}
int Pickup::gotIt()
{
    m_Spawned = false;
    m_SecondsSinceDeSpawn = 0;
    return m_Value;
}
void Pickup::upgrade()
{
    if (m_Type == 1)
    {
        m_Value += (HEALTH_START_VALUE * 0.5);
    }
    else
    {
        m_Value += (AMMO_START_VALUE * 0.5);
    }
    m_SecondsToLive += START_SECONDS_TO_LIVE / 10;
    m_SecondsToWait -= START_WAIT_TIME / 10;
}
```

Zombie Arena Game

```cpp
#include <SFML/Graphics.hpp>
#include <iostream>
#include "Player.cpp"
#include "zombie.cpp"
#include "bullet.cpp"
#include "pickup.cpp"
#include <sstream>
using namespace sf;
int createBackground(VertexArray &rVA, IntRect arena);
Zombie *createHorde(int numZombies, IntRect arena);
int main()
{
    int wave = 0;
    // The game will always be in one of four states
    enum class State
    {
        PAUSED,
        LEVELING_UP,
        GAME_OVER,
        PLAYING,
        NEXTWAVE
    };
    // Start with the GAME_OVER state
    State state = State::GAME_OVER;
    // Get the screen resolution and create an SFML window
    Vector2f resolution;
    // resolution.x = VideoMode::getDesktopMode().width;
    // resolution.y = VideoMode::getDesktopMode().height;
    resolution.x = 1920;
    resolution.y = 1080;
    RenderWindow window(VideoMode(800, 600),
                        "Zombie Arena", Style::Fullscreen);

    // Create a an SFML View for the main action
    View mainView(sf::FloatRect(0, 0, resolution.x, resolution.y));
    window.setView(mainView);
    int numZombies;
    int numZombiesAlive;
    Zombie *zombies = nullptr;

    Bullet bullets[100];
    int currentBullet = 0;
    int bulletsSpare = 24;
    int bulletsInClip = 6;
    int clipSize = 6;
    float fireRate = 3;
```

```
Time lastPressed;
// Here is our clock for timing everything
Clock clock;
// How long has the PLAYING state been active
Time gameTimeTotal;

// Where is the mouse in relation to world coordinates
Vector2f mouseWorldPosition;
// Where is the mouse in relation to screen coordinates
Vector2i mouseScreenPosition;
window.setMouseCursorVisible(true);
Texture textureCrosshair;
textureCrosshair.loadFromFile("crosshair.png");
Sprite spriteCrosshair;
spriteCrosshair.setTexture(textureCrosshair);
spriteCrosshair.setOrigin(25, 25);

// Create an instance of the Player class
Player player;
// The boundaries of the arena
IntRect arena;
arena.height = 500;
arena.width = 500;

// Create the background
VertexArray background;
// Load the texture for our background vertex array
Texture textureBackground;
textureBackground.loadFromFile("background_sheet.png");

int score = 0;
int hiScore = 0;
Pickup healthPickup(1);
Pickup ammoPickup(2);
// For the home/game over screen
Sprite spriteGameOver;
Texture textureGameOver;
textureGameOver.loadFromFile("background.png");
spriteGameOver.setTexture(textureGameOver);
spriteGameOver.setPosition(0, 0);

// Create a view for the HUD
View hudView(sf::FloatRect(0, 0, resolution.x, resolution.y));
View gameOverView(sf::FloatRect(0, 0, resolution.x, resolution.y));

// Create a sprite for the ammo icon
Sprite spriteAmmoIcon;
Texture textureAmmoIcon;
textureAmmoIcon.loadFromFile("ammo_icon.png");
```

```
    spriteAmmoIcon.setTexture(textureAmmoIcon);
    spriteAmmoIcon.setPosition(20, 980);

    // Load the font
    Font font;
    font.loadFromFile("zombiecontrol.ttf");

    // Paused
    Text pausedText;
    pausedText.setFont(font);
    pausedText.setCharacterSize(155);
    pausedText.setFillColor(Color::White);
    pausedText.setPosition(400, 400);
    pausedText.setString("Press Enter \nto continue");

    // Game Over
    Text gameOverText;
    gameOverText.setFont(font);
    gameOverText.setCharacterSize(125);
    gameOverText.setFillColor(Color::White);
    gameOverText.setString("Press Enter to play");
    FloatRect gameOverTextRect = gameOverText.getLocalBounds();
    gameOverText.setOrigin(gameOverTextRect.width / 2,
gameOverTextRect.height / 2);
    gameOverText.setPosition(gameOverView.getSize().x / 2,
gameOverView.getSize().y / 2);

    // Levelling up
    Text levelUpText;
    levelUpText.setFont(font);
    levelUpText.setCharacterSize(80);
    levelUpText.setFillColor(Color::White);
    levelUpText.setPosition(150, 250);
    std::stringstream levelUpStream;
    levelUpStream << "0- Start the normal game"
                  << "\n1- Increased rate of fire"
                  << "\n2- Increased clip size(next reload)"
                  << "\n3- Increased max health"
                  << "\n4- Increased run speed"
                  << "\n5- More and better health pickups"
                  << "\n6- More and better ammo pickups";
    levelUpText.setString(levelUpStream.str());

    // Ammo
    Text ammoText;
    ammoText.setFont(font);
    ammoText.setCharacterSize(55);
    ammoText.setFillColor(Color::White);
    ammoText.setPosition(200, 980);
```

```
    // Score
    Text scoreText;
    scoreText.setFont(font);
    scoreText.setCharacterSize(55);
    scoreText.setFillColor(Color::White);
    scoreText.setPosition(20, 0);

    // Hi Score
    Text hiScoreText;
    hiScoreText.setFont(font);
    hiScoreText.setCharacterSize(55);
    hiScoreText.setFillColor(Color::White);
    hiScoreText.setPosition(1400, 0);
    std::stringstream s;
    s << "Hi Score:" << hiScore;
    hiScoreText.setString(s.str());

    // Zombies remaining
    Text zombiesRemainingText;
    zombiesRemainingText.setFont(font);
    zombiesRemainingText.setCharacterSize(55);
    zombiesRemainingText.setFillColor(Color::White);
    zombiesRemainingText.setPosition(1500, 980);
    zombiesRemainingText.setString("Zombies: 100");

    Text waveNumberText;
    waveNumberText.setFont(font);
    waveNumberText.setCharacterSize(55);
    waveNumberText.setFillColor(Color::White);
    waveNumberText.setPosition(1250, 980);
    waveNumberText.setString("Wave: 0");

    // Health bar
    RectangleShape healthBar;
    healthBar.setFillColor(Color::Red);
    healthBar.setPosition(450, 980);

    // When did we last update the HUD?
    int framesSinceLastHUDUpdate = 0;

    // How often (in frames) should we update the HUD
    int fpsMeasurementFrameInterval = 1000;
    // The main game loop
    while (window.isOpen())
    {
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::KeyPressed)
            {
```

```
                // Pause a game while playing
                if (event.key.code == Keyboard::Return && state ==
State::PLAYING)
                {
                    state = State::PAUSED;
                }
                else if (event.key.code == Keyboard::Return && state ==
State::PAUSED)
                {
                    state = State::PLAYING;
                    clock.restart();
                }
                else if (event.key.code == Keyboard::Return && state ==
State::GAME_OVER)
                {
                    state = State::LEVELING_UP;
                    wave = 0;
                    currentBullet = 0;
                    bulletsSpare = 24;
                    bulletsInClip = 6;
                    clipSize = 6;
                    fireRate = 1;
                    score = 0;
                    player.resetPlayerStats();
                }

                if (state == State::PLAYING)
                {
                    if (event.key.code == Keyboard::R)
                    {
                        if (bulletsSpare >= clipSize)
                        {
                            // Plenty of bullets. Reload.
                            bulletsInClip = clipSize;
                            bulletsSpare -= clipSize;
                        }
                        else if (bulletsSpare > 0)
                        {
                            // Only few bullets left
                            bulletsInClip = bulletsSpare;
                            bulletsSpare = 0;
                        }
                        else
                        {
                            // More here soon?!
                        }
                    }
                }
            }
            // Handle the levelling up state
```

```
            if (state == State::LEVELING_UP)
            {
                // Handle the player levelling up
                if (event.key.code == Keyboard::Num0)
                {
                    state = State::NEXTWAVE;
                }
                if (event.key.code == Keyboard::Num1)
                {
                    fireRate++;
                    state = State::NEXTWAVE;
                }

                if (event.key.code == Keyboard::Num2)
                {
                    clipSize += clipSize;
                    state = State::NEXTWAVE;
                }

                if (event.key.code == Keyboard::Num3)
                {
                    player.upgradeHealth();
                    state = State::NEXTWAVE;
                }

                if (event.key.code == Keyboard::Num4)
                {
                    player.upgradeSpeed();
                    state = State::NEXTWAVE;
                }

                if (event.key.code == Keyboard::Num5)
                {
                    healthPickup.upgrade();
                    state = State::NEXTWAVE;
                }

                if (event.key.code == Keyboard::Num6)
                {
                    ammoPickup.upgrade();
                    state = State::NEXTWAVE;
                }
            } // End levelling up
        } // End event polling

        // Handle the player quitting
        if (Keyboard::isKeyPressed(Keyboard::Escape))
        {
            window.close();
        }
```

```
        if (state == State::NEXTWAVE)
        {
            wave++;
            arena.width = arena.width + (wave-1) * 100;
            arena.height = arena.height + (wave-1) * 100;
            arena.left = 0;
            arena.top = 0;

            // Pass the vertex array by reference
            // to the createBackground function
            int tileSize = createBackground(background, arena);

            // Spawn the player in the middle of the arena
            player.spawn(arena, resolution, tileSize);
            numZombies = wave * 2;
            delete[] zombies;
            zombies = createHorde(numZombies, arena);
            numZombiesAlive = numZombies;
            healthPickup.setArena(arena);
            ammoPickup.setArena(arena);
            // Reset the clock so there isn't a frame jump
            clock.restart();
            state = State::PLAYING;
        }
        // Handle controls while playing
        if (state == State::PLAYING)
        {
            // Handle the pressing and releasing of the WASD keys
            if (Keyboard::isKeyPressed(Keyboard::W))
            {
                player.moveUp();
            }
            else
            {
                player.stopUp();
            }

            if (Keyboard::isKeyPressed(Keyboard::S))
            {
                player.moveDown();
            }
            else
            {
                player.stopDown();
            }

            if (Keyboard::isKeyPressed(Keyboard::A))
            {
                player.moveLeft();
```

```
            }
            else
            {
                player.stopLeft();
            }

            if (Keyboard::isKeyPressed(Keyboard::D))
            {
                player.moveRight();
            }
            else
            {
                player.stopRight();
            }
            // Fire a bullet
            if (sf::Mouse::isButtonPressed(sf::Mouse::Left))
            {

                if (gameTimeTotal.asMilliseconds() -
lastPressed.asMilliseconds() > 1000 / fireRate && bulletsInClip > 0)
                {

                    // Pass the centre of the player
                    // and the centre of the cross-hair
                    // to the shoot function
                    bullets[currentBullet].shoot(
                        player.getCenter().x, player.getCenter().y,
                        mouseWorldPosition.x, mouseWorldPosition.y);

                    currentBullet++;
                    if (currentBullet > 99)
                    {
                        currentBullet = 0;
                    }
                    lastPressed = gameTimeTotal;

                    bulletsInClip--;
                }
            }

        } // End WASD while playing

        /*
        ****************
        UPDATE THE FRAME
        ****************
        */
        if (state == State::PLAYING)
        {
            // Update the delta time
```

```
            Time dt = clock.restart();
            // Update the total game time
            gameTimeTotal += dt;
            // Make a decimal fraction of 1 from the delta time
            float dtAsSeconds = dt.asSeconds();

            // Where is the mouse pointer
            mouseScreenPosition = Mouse::getPosition();

            // Convert mouse position to world coordinates of mainView
            mouseWorldPosition = window.mapPixelToCoords(
                Mouse::getPosition(), mainView);
            spriteCrosshair.setPosition(mouseWorldPosition);

            // Update the player
            player.update(dtAsSeconds, Mouse::getPosition());

            // Make a note of the players new position
            Vector2f playerPosition(player.getCenter());

            // Make the view centre around the player
            mainView.setCenter(player.getCenter());
            for (int i = 0; i < numZombies; i++)
            {
                zombies[i].update(dtAsSeconds, playerPosition);
            }

            // Update any bullets that are in-flight
            for (int i = 0; i < 100; i++)
            {
                if (bullets[i].isInFlight())
                {
                    bullets[i].update(dtAsSeconds);
                }
            }
            // Pickup updates
            healthPickup.update(dtAsSeconds);
            ammoPickup.update(dtAsSeconds);
            // Collision
            for (int i = 0; i < 100; i++)
            {
                for (int j = 0; j < numZombies; j++)
                {
                    if (bullets[i].isInFlight() &&
                        zombies[j].isAlive())
                    {
                        if
(bullets[i].getPosition().intersects(zombies[j].getPosition()))
                        {
                            // Stop the bullet
```

```
                            bullets[i].stop();

                            // Register the hit and see if it was a kill
                            if (zombies[j].hit())
                            {
                                // Not just a hit but a kill too
                                score += 10;
                                if (score >= hiScore)
                                {
                                    hiScore = score;
                                }

                                numZombiesAlive--;

                                // When all the zombies are dead (again)
                                if (numZombiesAlive == 0)
                                {
                                    state = State::NEXTWAVE;
                                }
                            }
                        }
                    }
                }
                for (int i = 0; i < numZombies; i++)
                {
                    if
(player.getPosition().intersects(zombies[i].getPosition()) &&
zombies[i].isAlive())
                    {

                        if (player.hit(gameTimeTotal))
                        {
                            // More here later
                        }

                        if (player.getHealth() <= 0)
                        {
                            state = State::GAME_OVER;
                        }
                    }
                }
                if
(player.getPosition().intersects(healthPickup.getPosition()) &&
healthPickup.isSpawned())
                {
                    player.increaseHealthLevel(healthPickup.gotIt());
                }

                // Has the player touched ammo pickup
```

```
            if (player.getPosition().intersects(ammoPickup.getPosition())
&& ammoPickup.isSpawned())
            {
                bulletsSpare += ammoPickup.gotIt();
            }
            // size up the health bar
            healthBar.setSize(Vector2f(player.getHealth() * 3, 70));

            // Update game HUD text
            std::stringstream ssAmmo;
            std::stringstream ssScore;
            std::stringstream ssHiScore;
            std::stringstream ssWave;
            std::stringstream ssZombiesAlive;

            // Update the ammo text
            ssAmmo << bulletsInClip << "/" << bulletsSpare;
            ammoText.setString(ssAmmo.str());

            // Update the score text
            ssScore << "Score:" << score;
            scoreText.setString(ssScore.str());

            // Update the high score text
            ssHiScore << "Hi Score:" << hiScore;
            hiScoreText.setString(ssHiScore.str());

            // Update the wave
            ssWave << "Wave:" << wave;
            waveNumberText.setString(ssWave.str());

            // Update the high score text
            ssZombiesAlive << "Zombies:" << numZombiesAlive;
            zombiesRemainingText.setString(ssZombiesAlive.str());

        } // End updating the scene

        /*
        **************
        Draw the scene
        **************
        */

        window.clear();
        window.setMouseCursorVisible(false);
        if (state == State::PLAYING)
        {

            // set the mainView to be displayed in the window
            // And draw everything related to it
```

```
            window.setView(mainView);

            // Draw the background
            // mainView.setCenter(Vector2f(arena.width/2,
arena.height/2));
            window.draw(background, &textureBackground);
            for (int i = 0; i < 100; i++)
            {
                if (bullets[i].isInFlight())
                {
                    window.draw(bullets[i].getShape());
                }
            }
            // Draw the player
            window.draw(player.getSprite());
            for (int i = 0; i < numZombies; i++)
            {
                window.draw(zombies[i].getSprite());
            }
            window.draw(spriteCrosshair);

            // Draw pickup
            if (ammoPickup.isSpawned())
            {
                window.draw(ammoPickup.getSprite());
            }
            if (healthPickup.isSpawned())
            {
                window.draw(healthPickup.getSprite());
            }
            window.setView(hudView);
            window.draw(healthBar);
            window.draw(spriteAmmoIcon);
            window.draw(ammoText);
            window.draw(scoreText);
            window.draw(hiScoreText);
            window.draw(waveNumberText);
            window.draw(zombiesRemainingText);
        } // end of draw while playing

        if (state == State::LEVELING_UP)
        {
            window.setView(gameOverView);
            window.draw(spriteGameOver);
            window.draw(levelUpText);
        }

        if (state == State::PAUSED)
        {
            window.setView(gameOverView);
```

```
            window.draw(pausedText);
        }

        if (state == State::GAME_OVER)
        {
            window.setView(gameOverView);
            window.draw(spriteGameOver);
            window.draw(gameOverText);
        }

        window.display();

    } // End game loop

    return 0;
}
int createBackground(VertexArray &rVA, IntRect arena)
{
    // Anything we do to rVA we are actually doing to background (in the
main function)

    // How big is each tile/texture
    const int TILE_SIZE = 50;
    const int TILE_TYPES = 3;
    const int VERTS_IN_QUAD = 4;

    int worldWidth = arena.width / TILE_SIZE;
    int worldHeight = arena.height / TILE_SIZE;

    // What type of primitive are we using?
    rVA.setPrimitiveType(Quads);

    // Set the size of the vertex array
    rVA.resize(worldWidth * worldHeight * VERTS_IN_QUAD);

    // Start at the beginning of the vertex array
    int currentVertex = 0;
    for (int w = 0; w < worldWidth; w++)
    {
        for (int h = 0; h < worldHeight; h++)
        {
            // Position each vertex in the current quad
            rVA[currentVertex + 0].position = Vector2f(w * TILE_SIZE, h *
TILE_SIZE);
            rVA[currentVertex + 1].position = Vector2f((w * TILE_SIZE) +
TILE_SIZE, h * TILE_SIZE);
            rVA[currentVertex + 2].position = Vector2f((w * TILE_SIZE) +
TILE_SIZE, (h * TILE_SIZE) + TILE_SIZE);
            rVA[currentVertex + 3].position = Vector2f((w * TILE_SIZE),
(h * TILE_SIZE) + TILE_SIZE);
```

```
            // Define the position in the Texture to draw for current
quad
            // Either mud, stone, grass or wall
            if (h == 0 || h == worldHeight - 1 || w == 0 || w ==
worldWidth - 1)
            {
                // Use the wall texture
                rVA[currentVertex + 0].texCoords = Vector2f(0, 0 +
TILE_TYPES * TILE_SIZE);
                rVA[currentVertex + 1].texCoords = Vector2f(TILE_SIZE, 0
+ TILE_TYPES * TILE_SIZE);
                rVA[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
TILE_SIZE + TILE_TYPES * TILE_SIZE);
                rVA[currentVertex + 3].texCoords = Vector2f(0, TILE_SIZE
+ TILE_TYPES * TILE_SIZE);
            }
            else
            {
                // Use a random floor texture
                srand((int)time(0) + h * w - h);
                int mOrG = (rand() % TILE_TYPES);
                int verticalOffset = mOrG * TILE_SIZE;

                rVA[currentVertex + 0].texCoords = Vector2f(0, 0 +
verticalOffset);
                rVA[currentVertex + 1].texCoords = Vector2f(TILE_SIZE, 0
+ verticalOffset);
                rVA[currentVertex + 2].texCoords = Vector2f(TILE_SIZE,
TILE_SIZE + verticalOffset);
                rVA[currentVertex + 3].texCoords = Vector2f(0, TILE_SIZE
+ verticalOffset);
            }

            // Position ready for the next for vertices
            currentVertex = currentVertex + VERTS_IN_QUAD;
        }
    }

    return TILE_SIZE;
}

Zombie *createHorde(int numZombies, IntRect arena)
{
    Zombie *zombies = new Zombie[numZombies];
    int maxX = arena.width - 20;
    int minX = arena.left + 20;
    int maxY = arena.height - 20;
    int minY = arena.top + 20;
```

```
    for (int i = 0; i < numZombies; i++)
    {
        srand((int)time(0) * i);
        int side = (rand() % 4);
        float x, y;
        switch (side)
        {
        case 0:
            x = minX;
            y = (rand() % maxY) + minY;
            break;
        case 1:
            x = maxX;
            y = (rand() % maxY) + minY;
            break;
        case 2:
            y = minY;
            x = (rand() % maxX) + minX;
            break;
        case 3:
            y = maxY;
            x = (rand() % maxX) + minX;
            break;
        }
        srand((int)time(0) * i * 2);
        int type = (rand() % 3);
        zombies[i].spawn(x, y, type, i);
    }
    return zombies;
}
```

Program Collection

Collection of Programming Examples

WAP to print `Hello World!!` on the screen using C++.

```
#include<iostream>
int main()
{
    std::cout << "Hello World!!" << std::endl;
    return(0);
}
```

or

```
#include<iostream>
using namespace std;
int main()
{
    cout << "Hello World!!" << endl;
    return(0);
}
```

WAP to find LCM two given number using recursion

```
#include <iostream>
using namespace std;
int lcm(int m, int n, int a)
{
    if((a%m == 0) && (a%n ==0))
        return(a);
    else
        return(lcm(m,n, a+n));
}
int main()
{
    int m, n;
    cout << "Enter the 1st Num :";
    cin >> m;
    cout << "Enter the 2nd Num :";
    cin >> n;
    cout << "LCM of " << m << " & " << n << "is : " << lcm(m,n,n) <<
endl;
    return(0);
}
```

WAP to get 5 student details using structure and print on the screen

```cpp
//WAP to get 5 student details using structure and print on the screen

#include<iostream>
#include<iomanip>
using namespace std;
struct student
{
    char name[30];
    int roll;
    float cgpa;
};
int main()
{
    struct student s[5];
    for(int i=0;i<5;i++)
    {
        cout << "Name of " << i << "th student : ";
        if(i!=0)
            cin.ignore();
        cin.getline(s[i].name, 30);
        cout << "Roll of " << i << "th student : ";
        cin >> s[i].roll;
        cout << "cgpa of " << i << "th student : ";
        cin >> s[i].cgpa;
    }

    // Print header with column titles
    cout << left << setw(30) << "Name"    // Left align, width 30
         << setw(10) << "Roll No"        // Left align, width 10
         << setw(10) << "cgpa"          // Left align, width 10
         << endl;

    // Print a separator line
    cout << setfill('-') << setw(50) << "-" << endl;
    cout << setfill(' '); // Reset fill to space

    // Print data in columns
    for (int i = 0; i < 5; i++) {
        cout << left << setw(30) << s[i].name
             << setw(10) << s[i].roll
             << fixed << setprecision(2) << setw(10) << s[i].cgpa
             << endl;
    }
    return(0);
}
```

Write a code to create a student class and provide appropriate functions to set, display and modify cgpa with a public driver function.

```
#include<iostream>
using namespace std;

class student{
    int id;
    char name[30];
    float cgpa;
    void getdata()
    {
        cout << "Eneter id : ";
        cin >> id;
        cout << "Enter name : ";
        cin.ignore();
        cin.getline(name, 30);
        cout << "Enter the CGPA : ";
        cin >> cgpa;
    }
    void printdata()
    {
        cout << "Name : " << name << endl;
        cout << "ID : " << id << endl;
        cout << "CGPA : " << cgpa << endl;
    }
    void changecgpa()
    {
        cout << "Enter new CGPA : ";
        cin >> cgpa;
    }
public:
    void driver()
    {
        int i;
        cout << "Enter 1 : for entring information " << endl;
        cout << "Enter 2 : for showing information " << endl;
        cout << "Enter 3 : for changing cgpa value " << endl;
        cout << "enter your choice :";
        cin >> i;
        switch(i)
        {
            case 1: getdata(); break;
            case 2: printdata(); break;
            case 3: changecgpa(); break;
            default: cout << "Wrong Choice ...." << endl;
        }
```

```
    }
};
int main()
{
    student s1;
    for(int i=0;i<4;i++)
        s1.driver();
    return(0);
}
```

Write a code to use the above student class to store and display details of `n` numbers of students.

```
#include<iostream>
using namespace std;

class student{
    int id;
    char name[30];
    float cgpa;
    void getdata()
    {
        cout << "Eneter id : ";
        cin >> id;
        cout << "Enter name : ";
        cin.ignore();
        cin.getline(name, 30);
        cout << "Enter the CGPA : ";
        cin >> cgpa;
    }
    void printdata()
    {
        cout << "Name : " << name << endl;
        cout << "ID : " << id << endl;
        cout << "CGPA : " << cgpa << endl;
    }
    void changecgpa()
    {
        cout << "Enter new CGPA : ";
        cin >> cgpa;
    }
public:
    void driver(int i)
    {
        switch(i)
        {
            case 1: getdata(); break;
            case 2: printdata(); break;
```

```
            case 3: changecgpa(); break;
            default: cout << "Wrong Choice ...." << endl;
        }
    }
};
int main()
{
    int n;
    cout >> "Enter the number of students : ";
    cin << n;
    student s[n];
    // Store the details
    for(int i=0;i<n;i++)
        s[i].driver(1);
    // Show the details
    for(int i=0;i<n;i++)
        s[i].driver(2);
    return(0);
}
```

Write a program in C++ that simulates an online shopping `cart` system. Create a structure `item` having data members as `ItemName`, `ItemCost`, `ItemCount`. The Cart class needs item count from the user and uses item structure to store information. Use appropriate functions to set, print and calculate the total cost of the items.

```
#include<iostream>
using namespace std;

struct item{
    char ItemName[20];
    float ItemCost;
    int ItemCount;
};

class cart
{
private:
    int n;
    item *itm;
    float TotalCost;
public:
    void getdata()
    {
        cout << "Enter the number of item :";
```

```
        cin >> n;
        itm = new item[n];
    }
    void getItemDetails()
    {
        for(int i=0;i<n;i++)
        {
            cout << "Item Name : ";
            cin.ignore();
            cin.getline(itm[i].ItemName,20);
            cout << "Item Cost : ";
            cin >> itm[i].ItemCost;
            cout << "Item count : ";
            cin >> itm[i].ItemCount;
        }
    }
    void calculatePrice()
    {
        TotalCost = 0;
        for(int i=0;i<n;i++)
        {
            TotalCost = TotalCost + (itm[i].ItemCost * itm[i].ItemCount);
        }
        cout << "Total cost : " << TotalCost << endl;
    }
};
int main()
{
    cart c1;
    c1.getdata();
    c1.getItemDetails();
    c1.calculatePrice();
    return(0);
}
```

WAP for banking applications where each time a new account is created, the account holder's details and initial balance must be initialized using a constructor and when an account object goes out of scope, it should display a message saying the account is closed using a destructor.

```
#include <iostream>
#include <cstring>
using namespace std;

class BankAccount {
private:
```

```
    char accountHolder[30];
    int accountNumber;
    double balance;

public:
    // Defining default constructor
    BankAccount()
    {
        cout << "Acount creation process stared ... "<<endl;
    }
    // Constructor to initialize account details
    BankAccount(char *name, int accNum, double initialBalance) {
        strcpy(accountHolder,name);
        accountNumber = accNum;
        balance = initialBalance;
        cout << "Account created successfully!" << endl;
        cout << "Account Holder: " << accountHolder << endl;
        cout << "Account Number: " << accountNumber << endl;
        cout << "Initial Balance: " << balance << endl;
    }

    // Destructor to display message when object goes out of scope
    ~BankAccount() {
        cout << "Account " << accountNumber
            << " belonging to " << accountHolder
            << " is closed." << endl;
    }

    //void get data from the user
    void get_data()
    {
        cout << "Account Holder: " << endl;
        cin.ignore();
        cin.getline(accountHolder, 30);
        cout << "Account Number: " << endl;
        cin >> accountNumber;
        cout << "Initial Balance: " << endl;
        cin >> balance;
    }

    // Function to deposit money
    void deposit(double amount) {
        balance += amount;
        cout << "Deposited: " << amount
            << " | New Balance: " << balance
            << endl;
    }

    // Function to withdraw money
    void withdraw(double amount) {
```

```
        if (amount > balance) {
            cout << "Insufficient balance!" << endl;
        } else {
            balance -= amount;
            cout << "Withdrawn: " << amount
                    << " | New Balance: " << balance
                    << endl;
        }
    }

    // Function to display account details
    void display() {
        cout << "Account Holder: " << accountHolder << endl;
        cout << "Account Number: " << accountNumber << endl;
        cout << "Balance: " << balance << endl;
    }
};

int main() {
    // Creating an account
    BankAccount acc("Dibyasundar Das", 101, (double)(5000.0));
    acc.deposit(2000);
    acc.withdraw(1500);
    acc.display();
    // Account goes out of scope here, triggering destructor
    return 0;
}
```

For a library, you want to keep track of books being borrowed and returned. When a book is borrowed, its title, author, and borrower name are recorded. When the book is returned (object destroyed), display a message with the borrower name and book title.

```
#include <iostream>
#include <cstring>
using namespace std;

class LibraryBook {
private:
    char title[100];
    char author[100];
    char borrower[100];

public:
    // Constructor to initialize book details
    LibraryBook(const char* bookTitle, const char* bookAuthor, const
```

```
char* borrowerName) {
        strcpy(title, bookTitle);
        strcpy(author, bookAuthor);
        strcpy(borrower, borrowerName);

        cout << "Book Borrowed: " << title
            << " by " << author
            << " | Borrower: " << borrower
            << endl;
    }

    // Destructor to display message when book is returned
    ~LibraryBook() {
        cout << "Book Returned: " << title
            << " | Borrower: " << borrower
            << endl;
    }
};

int main() {
    int choice;
    LibraryBook *book1;
    while(1)
    {
        cout << "Press 1 : Issue a book "<<endl;
        cout << "Press 2 : Return a book "<<endl;
        cin >> choice;
        // Borrowing a book
        if (choice ==1)
        {
            book1 = new LibraryBook("Wings of Fire",
                    " Arun Tiwari", "Dibyasundar Das");
        }
        else
        {
            delete book1;
        }
    }

    return 0;
}
```

WAP to Display a window of size 800 × 600

```
#include<SFML/Graphics.hpp>
using namespace sf;

inr main()
```

```
{
    VideoMode video(800,600);
    RenderWindow window(video, "Window!");

    while(window.isOpen())
    {
        window.clear();
        window.display();
    }
}
```

Write a code snippet to close the window when close button is pressed.

```
while(window.isOpen())
{
    Event ev;
    while(window.pollEvent(ev))
    {
        if(ev.type == Event::closed)
        {
            window.close();
        }
    }
}
```

Write a code snippet to close the window when escape key is pressed.

```
while(window.isOpen())
{
    Event ev;
    while(window.pollEvent(ev))
    {
        if(ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::Escape)
        {
            window.close();
        }
    }
}
```

OR

```
while(window.isOpen())
{
    if(Keyboard::isKeyPressed(Keyboard::Escape))
    {
        window.close();
    }
}
```

GPWC Quizzes

Quiz 001 2025 6th Sem (Sec : 2241001)

1. **Which of the following is the correct syntax for a class in C++?**
   A. `class MyClass { int a; };`
   B. `MyClass { class int a; };`
   C. `def class MyClass { int a; };`
   D. `create MyClass { int a; };`

Answer: A. `class MyClass { int a; };`

2. What is `cout` ?
   A. It is a function
   B. It is a class
   C. It is an object (class instance)
   D. It is a reserved word (C++ keyword)

Answer: C. It is an object (class instance)

3. What is the output of `cout << (3 == 3);` ?
   A. true
   B. false
   C. 0
   D. 1

Answer: D. 1

4.

```
void swap (int &a, int &b) {
     int temp; temp = a;
     a = b;
     b = temp; }
int main () {
     int i = 0, j = 1;
     swap (i, j);
     cout << i << " " << j << endl;
 }
```

What is the output of the code:
A. 0 1
B. 1 0

C. Syntax Error
D. Runtime Error

Answer: B. 1 0

5.

```
void myfun(int i, int &k) {
    i = 1;
    k = 2;
}
int main () {
    int x = 0;
    myfun (x, x);
    cout << x << endl;
    return 0;
}
```

What is the output of the code:
A. 0
B. 1
C. 2
D. None of these

Answer: C. 2

6.

```
class MyClass{
public:
    ~MyClass() {
        cout<<"My destructor"<<endl;
    }
};

void main()
{
    MyClass obj;
    obj.~MyClass();
}
```

What is the output of the code:
A. My destructor
B.
My destructor

My destructor
C. Error
D. None Output

Answer: B.
My destructor
My destructor

7.

```
class Sample {
public:
    Sample(int x = 10) {
        cout << "Value: " << x << endl;
    }
};
int main() {
    Sample obj;
    return 0;
}
```

What will be the output?
A. Compilation error
B. Value: 10
C. No output
D. Value: 0

Answer: B. Value: 10

8. **What is a default constructor in C++?**
   A. A constructor that takes at least one argument
   B. A constructor that does not take any arguments
   C. A constructor with a default argument value
   D. A constructor that initializes only static members

Answer: B. A constructor that does not take any arguments

9. **Which feature in OOP allows reusing code?**
   A. Polymorphism
   B. Inheritance
   C. Encapsulation
   D. Data hiding

Answer: B. Inheritance

10. **Null character needs a space of**
    A. zero bytes
    B. one byte
    C. three bytes
    D. four bytes

Answer: B. one byte

11.

```
void Cube(double  y) {
   y = y*y*y;
}
void main()
{
     double g = 4.0;
     Cube(g);
     cout << g<<endl;
     return(0);
}
```

Find the output of the above code
A. 4
B. 64
C. 0
D. Non-of the above

Answer: A. 4

12.

```
void Cube(double  &y) {
   y = y*y*y;
}
void main()
{
     double g = 4.0;
     Cube(g);
     cout << g<<endl;
     return(0);
}
```

Find the output of the above code

A. 4

B. 64

C. 0

D. Non-of the above

Answer: B. 64

13. Which of the following operator is overloaded for object `cout` ?
    A. >>
    B. <<
    C. +
    D. =

Answer: A. >>

14. **What is the purpose of the scope resolution operator (: :) in C++?**
    A. To define a global variable inside a function
    B. To resolve the scope of a variable or function
    C. To create an object of a class
    D. To include a header file

Answer: B. To resolve the scope of a variable or function

15. **When struct is used instead of the keyword class means, what will happen in the program?**
    A. access is public by default
    B. access is private by default
    C. access is protected by default
    D. none of the mentioned

Answer: A. access is public by default*

Quiz 001 2025 6th Sem (Sec : 2241002)

1. **What will be the output of `cout << (5 / 2);` ?**
   A. 2.5
   B. 2
   C. 3
   D. 2.0

Answer: B. 2

2. Which operator is used to allocate memory dynamically in C++?
   A. malloc
   B. new
   C. alloc
   D. create

Answer: B. new

3. What is the output of `cout << (3 == 3);`?
   A. true
   B. false
   C. 0
   D. 1

Answer: D. 1

4.

```
int main() {
    float a = 5, b = 2;
    cout << a / b;
    return 0;
}
```

What is the output of the code:
A. 2.5
B. 2
C. 3
D. Compilation error

Answer: A. 2.5

5.

```
int main() {
    int x = 10;
    int &y = x;
    y = 20;
    cout << x;
    return 0;
}
```

What is the output of the code:
A. 10

B. 20
C. Compilation error
D. Garbage value

Answer: B. 20

6.

```
int main() {
    int x = 5;
    cout << (x << 1);
    return 0;
}
```

What is the output of the code:
A. 2
B. 10
C. 5
D. Compilation error

Answer: B. 10

7.

```
class Test {
public:
    int x;
};

int main() {
    Test obj;
    cout << obj.x;
    return 0;
}
```

What is the output of the code:
A. 0
B. Garbage value
C. Compilation error
D. Runtime error

Answer: B. Garbage value (Uninitialized variable)

8.

```
class Sample {
public:
    Sample(int x = 10) {
        cout << "Value: " << x << endl;
    }
};
int main() {
    Sample obj;
    return 0;
}
```

What will be the output?
A. Compilation error
B. Value: 10
C. No output
D. Value: 0

Answer: B. Value: 10

9. What is a default constructor in C++?
   A. A constructor that takes at least one argument
   B. A constructor that does not take any arguments
   C. A constructor with a default argument value
   D. A constructor that initializes only static members

Answer: B. A constructor that does not take any arguments

10. Which feature in OOP allows reusing code?
    A. Polymorphism
    B. Inheritance
    C. Encapsulation
    D. Data hiding

Answer: B. Inheritance

11.

```
class MyClass {
public:
    MyClass() {cout << "MyClass ";}
    ~MyClass() {cout << "~MyClass ";}
};
```

```
void myFunc() { MyClass obj;}

int main() {
    myFunc();
    cout << "Main ";
    return 0;
}
```

What will be the output?
A. MyClass Main ~MyClass
B. Main MyClass ~MyClass
C. MyClass ~MyClass Main
D. Compilation error

Answer: C. MyClass ~MyClass Main

12. Null character needs a space of
    A. zero bytes
    B. one byte
    C. three bytes
    D. four bytes

Answer: B. one byte

13.

```
void Cube(double  y) {
   y = y*y*y;
}
int main()
{
     double g = 4.0;
     Cube(g);
     cout << g<<endl;
     return(0);
}
```

Find the output of the above code
A. 4
B. 64
C. 0
D. Non-of the above

Answer: A. 4

14.

```
void Cube(double  &y) {
   y = y*y*y;
}
int main()
{
     double g = 4.0;
     Cube(g);
     cout << g<<endl;
     return(0);
}
```

Find the output of the above code
A. 4
B. 64
C. 0
D. Non-of the above

Answer: B. 64

15.

```
class A {
public:
    A() { cout << "A "; }
    ~A() { cout << "~A "; }
};

void func() {
    static A obj;
}

int main() {
    func();
    func();
    cout << "Main ";
    return 0;
}
```

Find the output of the above code
A. A Main ~A
B. A A Main ~A

C. A Main ~A
D. Compilation error

Answer: C. A Main ~A

Can destructor be over loaded ?
A. True
B. False

Answer: B. False

Quiz 002 2025 6th Sem (Sec : 2241001)

WAP program to create a game with following objectives

1. There is a basket at bottom which can only move left and right.
2. There are 2 types of balls coming from the top to down in a straight line, namely Red Balls and White Balls.
3. The ratio of red to white ball is 6:1 where Red Ball are unsafe and White Ball as safe to collect.
4. The gamer has to collect White Ball while avoiding the Red Balls. Each successful collection will increase the score by 1.
5. There is a time bar which indicate the elapsed time. With every success in collecting the white ball the gamer will be rewarded with additional time.
6. Goal is to collect as many as white Ball Possible without touching Red Ball and before time runs out.
7. You have to implement 2 HUD
    1. Score counter
    2. Time Bar
8. Additionally Game should be paused and un-paused with space and the game can be restarted with Enter Key.

```cpp
#include <SFML/Graphics.hpp>
#include <iostream>
#include <sstream>

using namespace sf;
using namespace std;

int main()
{
    VideoMode video(800, 600);
    RenderWindow window(video, "Catch Ball Game");

    float windowWidth = 1980;
    float windowHeight = 1020;
    View view(FloatRect(0, 0, windowWidth, windowHeight));

    window.setView(view);

    RectangleShape batShape;
    float batWidth = 200;
    float batHeight = 10;
    float batSpeed = 2;
    float batPixSec = windowHeight / batSpeed;
```

```
    batShape.setOrigin(batWidth / 2, batHeight / 2);
    batShape.setSize(Vector2f(batWidth, batHeight));
    batShape.setFillColor(Color::White);
    batShape.setPosition(windowWidth / 2, windowHeight - 20);

    CircleShape ballShape;
    float radius = 20;
    float ballSpeed = 3;
    float ballPixSecX = windowWidth / ballSpeed;
    float ballPixSecY = windowHeight / ballSpeed;
    ballShape.setRadius(radius);
    ballShape.setFillColor(Color::Magenta);
    ballShape.setPosition(20 + rand() % (int)(windowWidth - 20), -100);

    CircleShape ballEnemyShape[6];
    for (int i = 0; i < 6; i++)
    {
        ballEnemyShape[i].setRadius(radius);
        ballEnemyShape[i].setFillColor(Color::Red);
        float x = 20 + rand() % (int)(windowWidth - 20);
        float y = -200 + rand() % 200;
        ballEnemyShape[i].setPosition(x, y);
    }

    Font font;
    font.loadFromFile("KOMIKAP_.ttf");

    Text scoreText;
    scoreText.setFont(font);
    scoreText.setCharacterSize(75);
    scoreText.setFillColor(Color::White);
    scoreText.setPosition(20, 20);

    Text liveText;
    liveText.setFont(font);
    liveText.setCharacterSize(75);
    liveText.setFillColor(Color::White);

    float scoreVal = 0;
    float liveVal = 3;

    bool paused = false;
    bool bounceTop = false;
    bool bounceBottom = true;
    bool gameOver = false;

    Clock ct;
    Time dt;

    while (window.isOpen())
```

```
    {
        dt = ct.restart();
        Event ev;
        while (window.pollEvent(ev))
        {
            if (ev.type == Event::Closed || (ev.type == Event::KeyPressed
&& ev.key.code == Keyboard::Escape))
                window.close();
            if (ev.type == Event::KeyPressed && ev.key.code ==
Keyboard::Enter && gameOver)
            {
                paused = false;
                gameOver = false;
                bounceTop = false;
                bounceBottom = true;
                scoreVal = 0;
                liveVal = 3;
                ballShape.setPosition(windowWidth / 2, 20);
            }
        }

        stringstream ss, ss1;
        ss << "Score : " << scoreVal;
        scoreText.setString(ss.str());

        if (!gameOver)
            ss1 << "Lives : " << liveVal;
        else
            ss1 << "Game Over .. Press Enter to restart";
        liveText.setString(ss1.str());

        FloatRect details = liveText.getLocalBounds();
        liveText.setPosition(1980 - details.width - 20, 20);

        if (!paused)
        {
            if (Keyboard::isKeyPressed(Keyboard::Left))
            {
                float x = batShape.getPosition().x;
                float y = batShape.getPosition().y;
                x = x - dt.asSeconds() * batPixSec;
                if (x < batWidth / 2)
                    x = batWidth / 2;
                batShape.setPosition(x, y);
            }
            if (Keyboard::isKeyPressed(Keyboard::Right))
            {
                float x = batShape.getPosition().x;
                float y = batShape.getPosition().y;
                x = x + dt.asSeconds() * batPixSec;
```

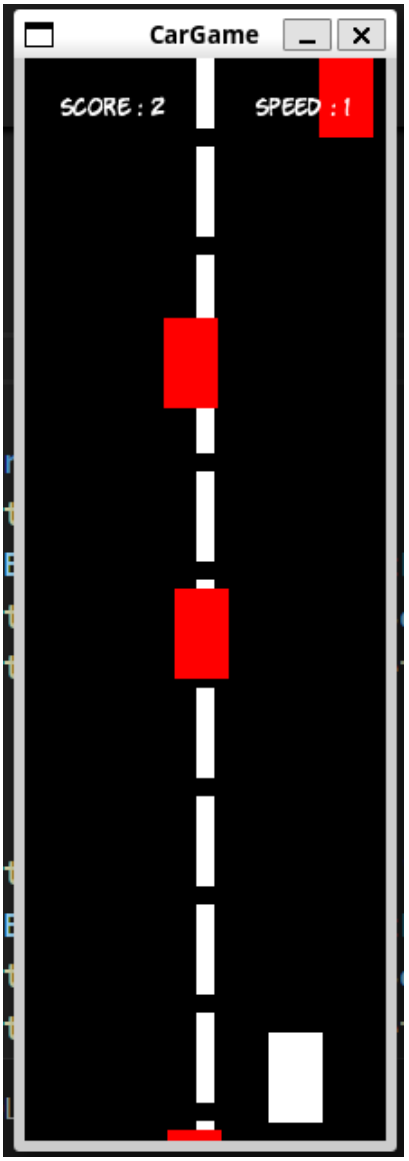
```
                if (x > windowWidth - batWidth / 2)
                    x = windowWidth - batWidth / 2;
                batShape.setPosition(x, y);
            }
            float x = ballShape.getPosition().x;
            float y = ballShape.getPosition().y;
            y = y + ballPixSecY * dt.asSeconds();
            if (y > view.getSize().y)
            {
                y = -40;
                x = 20 + rand() % (int)(view.getSize().x - 40);
            }
            ballShape.setPosition(x, y);

            for (int i = 0; i < 6; i++)
            {
                float x = ballEnemyShape[i].getPosition().x;
                float y = ballEnemyShape[i].getPosition().y;
                y = y + ballPixSecY * dt.asSeconds();
                if (y > view.getSize().y)
                {
                    y = -40;
                    x = 20 + rand() % (int)(view.getSize().x - 40);
                }
                ballEnemyShape[i].setPosition(x, y);
            }

            if
(ballShape.getGlobalBounds().intersects(batShape.getGlobalBounds()))
            {
                ballShape.setPosition(20 + rand() % (int)
(view.getSize().x - 40), -40);
                scoreVal ++;
            }
            for (int i = 0; i < 6; i++)
            {
                if
(ballEnemyShape[i].getGlobalBounds().intersects(batShape.getGlobalBounds(
)))
                {
                    ballEnemyShape[i].setPosition(20 + rand() % (int)
(view.getSize().x - 40), -40);
                    liveVal -- ;
                }
            }
        }


        if (liveVal == 0)
        {
```

```
            paused = true;
            gameOver = true;
            ballShape.setPosition(windowWidth / 2, windowHeight / 2);
        }

        window.clear();

        window.draw(batShape);
        window.draw(ballShape);
        for (int i = 0; i < 6; i++)
        {
            window.draw(ballEnemyShape[i]);
        }

        window.draw(scoreText);
        window.draw(liveText);

        window.display();
    }
}
```

Quiz 002 2025 6th Sem (Sec : 2241002)

WAP program to create a game with following objectives

1. Design and implement a 2D car racing game using C++ and the SFML (Simple and Fast Multimedia Library) framework.
2. In this game, the player will control a car that moves left and right on a vertically scrolling road, avoiding enemy cars that approach from the top of the screen.
3. The game should include real-time player input handling, enemy spawning, collision detection, scorekeeping, and a game-over scenario when a collision occurs.
4. Enemy cars spawn from the top and move downward. Avoid collisions with enemy car. Increase the speed of enemy cars as time progresses for difficulty.
5. The background should scroll to simulate forward motion, and the difficulty should increase over time by making enemy cars move faster.
6. The game must also display the current score on the screen, and after a collision, show a “Game Over” screen with the final score and an option to restart the game.



```
#include <SFML/Graphics.hpp>
#include <iostream>
#include <sstream>
using namespace sf;
using namespace std;

int main()
{
    VideoMode video(200, 600);
    RenderWindow window(video, "CarGame", Style::Titlebar |
Style::Close);

    RectangleShape divider[11];
    for (int i = 0; i < 11; i++)
    {
        divider[i].setSize(Vector2f(10, 50));
        divider[i].setPosition(95, -60 + (i * 60));
        divider[i].setFillColor(Color::White);
    }
    float dividerSpeed = 5;
    float diviserPixelPerSec = 600 / dividerSpeed;

    RectangleShape carDrive;
    carDrive.setSize(Vector2f(30, 50));
    carDrive.setFillColor(Color::White);
    carDrive.setPosition(40, 540);
    float carSpeed = 1;
    float carPixelPerSec = 200 / carSpeed;

    RectangleShape enemyCar[7];
    for (int i = 0; i < 7; i++)
    {
        enemyCar[i].setSize(Vector2f(30, 50));
        float x = rand() % 170;
        if (rand() % 3 != 0)
            enemyCar[i].setPosition(x, -900 + (i * 150));
        else
            enemyCar[i].setPosition(210, -900 + (i * 150));
        enemyCar[i].setFillColor(Color::Red);
    }
    float enemyCarPixelPerSec = 60;

    Clock ct;
    Time dt;

    bool gameOver = false;
    bool paused = false;
    bool acceptInput = true;
```

```
    int score = 0;
    int gameSpeed = 1;

    Text scoreHud;
    Text speedHud;
    Text message;

    Font ft;
    ft.loadFromFile("KOMIKAP_.ttf");

    scoreHud.setFont(ft);
    scoreHud.setFillColor(Color::White);
    scoreHud.setCharacterSize(12);
    scoreHud.setString("Score : 0");
    scoreHud.setPosition(20, 20);

    speedHud.setFont(ft);
    speedHud.setFillColor(Color::White);
    speedHud.setCharacterSize(12);
    speedHud.setString("Speed : 1");
    FloatRect speedHudBound = speedHud.getLocalBounds();
    speedHud.setPosition(window.getSize().x - 20 - speedHudBound.width,
20);

    message.setFont(ft);
    message.setFillColor(Color::White);
    message.setCharacterSize(20);
    message.setString("");
    FloatRect messageHudBound = message.getLocalBounds();
    message.setOrigin(messageHudBound.width / 2, messageHudBound.height /
2);
    message.setPosition(window.getSize().x / 2, window.getSize().y / 2);

    while (window.isOpen())
    {
        dt = ct.restart();
        Event event;
        while (window.pollEvent(event))
        {
            if (event.type == Event::Closed)
            {
                window.close();
            }
            if (event.type == Event::KeyReleased)
            {
                acceptInput = true;
            }
            if (event.type == Event::KeyPressed)
            {
                if (event.key.code == Keyboard::Enter && gameOver)
```

```
                {
                    paused = false;
                    gameOver = false;
                    score = 0;
                    enemyCarPixelPerSec = 60;
                    gameSpeed = 1;
                    for (int i = 0; i < 7; i++)
                    {
                        enemyCar[i].setSize(Vector2f(30, 50));
                        float x = rand() % 170;
                        if (rand() % 3 == 0)
                            enemyCar[i].setPosition(x, -900 + (i * 150));
                        else
                            enemyCar[i].setPosition(210, -900 + (i *
150));
                        enemyCar[i].setFillColor(Color::Red);
                    }
                    message.setString("");
                    messageHudBound = message.getLocalBounds();
                    message.setOrigin(messageHudBound.width / 2,
messageHudBound.height / 2);
                    message.setPosition(window.getSize().x / 2,
window.getSize().y / 2);
                    acceptInput = false;
                }
            }
        }
        if(Keyboard::isKeyPressed(Keyboard::Space) && acceptInput)
        {
            paused = !paused;
            acceptInput = false;
        }
        if (!paused)
        {
            if (Keyboard::isKeyPressed(Keyboard::Left))
            {
                float x = carDrive.getPosition().x;
                float y = carDrive.getPosition().y;
                x = x - dt.asSeconds() * carPixelPerSec;
                if (x < 0)
                    x = 0;

                carDrive.setPosition(x, y);
            }
            if (Keyboard::isKeyPressed(Keyboard::Right))
            {
                float x = carDrive.getPosition().x;
                float y = carDrive.getPosition().y;
                x = x + dt.asSeconds() * carPixelPerSec;
                if (x > 170)
```

```
                x = 170;

            carDrive.setPosition(x, y);
        }

        stringstream ss, ss1;
        ss << "Score : " << score;
        scoreHud.setString(ss.str());

        ss1 << "Speed : " << gameSpeed;
        speedHud.setString(ss1.str());

        for (int i = 0; i < 7; i++)
        {
            float x = enemyCar[i].getPosition().x;
            float y = enemyCar[i].getPosition().y;
            y = y + enemyCarPixelPerSec * dt.asSeconds();
            if (y > 600)
            {
                score = score + 1;
                if (score % 10 == 0)
                {
                    gameSpeed += 1;
                    enemyCarPixelPerSec = 60 + (gameSpeed * 10);
                }
                y = -450;
                if (rand() % 3 != 0)
                    x = rand() % 170;
                else
                    x = 210;
            }
            enemyCar[i].setPosition(x, y);
        }

        for (int i = 0; i < 11; i++)
        {
            float y = divider[i].getPosition().y;
            y = y + diviserPixelPerSec * dt.asSeconds();
            if (y > 600)
            {
                y = -60;
            }
            divider[i].setPosition(95, y);
        }

        for (int i = 0; i < 7; i++)
        {
            if
(enemyCar[i].getGlobalBounds().intersects(carDrive.getGlobalBounds()))
            {
```

```
                    gameOver = true;
                }
            }
        }


        if (gameOver)
        {
            paused = true;
            message.setString("Game Over !!!");
            messageHudBound = message.getLocalBounds();
            message.setOrigin(messageHudBound.width / 2,
messageHudBound.height / 2);
            message.setPosition(window.getSize().x / 2,
window.getSize().y / 2);
        }
        else if(paused && !gameOver)
        {
            message.setString("Paused !!!");
            messageHudBound = message.getLocalBounds();
            message.setOrigin(messageHudBound.width / 2,
messageHudBound.height / 2);
            message.setPosition(window.getSize().x / 2,
window.getSize().y / 2);
        }
        else
        {
            message.setString("");
            messageHudBound = message.getLocalBounds();
            message.setOrigin(messageHudBound.width / 2,
messageHudBound.height / 2);
            message.setPosition(window.getSize().x / 2,
window.getSize().y / 2);
        }

        window.clear();

        for (int i = 0; i < 11; i++)
        {
            window.draw(divider[i]);
        }
        for (int i = 0; i < 7; i++)
        {
            window.draw(enemyCar[i]);
        }

        window.draw(carDrive);

        window.draw(scoreHud);
        window.draw(speedHud);
```

```
        window.draw(message);

        window.display();
    }

    return (0);
}
```