**GAME 2001**

**Assignment #1 (To be done in Pairs)**

**Due Date: 7th October 2018 11:59pm (end of the day)**

**Date Given: 10th September 2018**

**Weight 20% (20 marks)**

This Assignment simulates a battle between two Carriers in space. Each Carrier has a number of fighters on board located in their bays. The information for each Carrier, as well as all their fighters, is located in a file ***shipData.txt***. A sample one has been uploaded along with the assignment. You are encouraged to make your own data file.

This assignment will require two classes named Carrier and Fighter to be made. The details for each class is given in the following pages. The main function will instantiate these classes and load the respective fighter objects into their carrier objects. After loading all the objects, the main function will call a function named “battle” with the two Carrier objects as parameters. The battle function will simulate the battle and print all output concerning the battle to the display

NOTE: All the classes and main function should be in one file called **Assignment1.cpp**.

**The CARRIER class:**

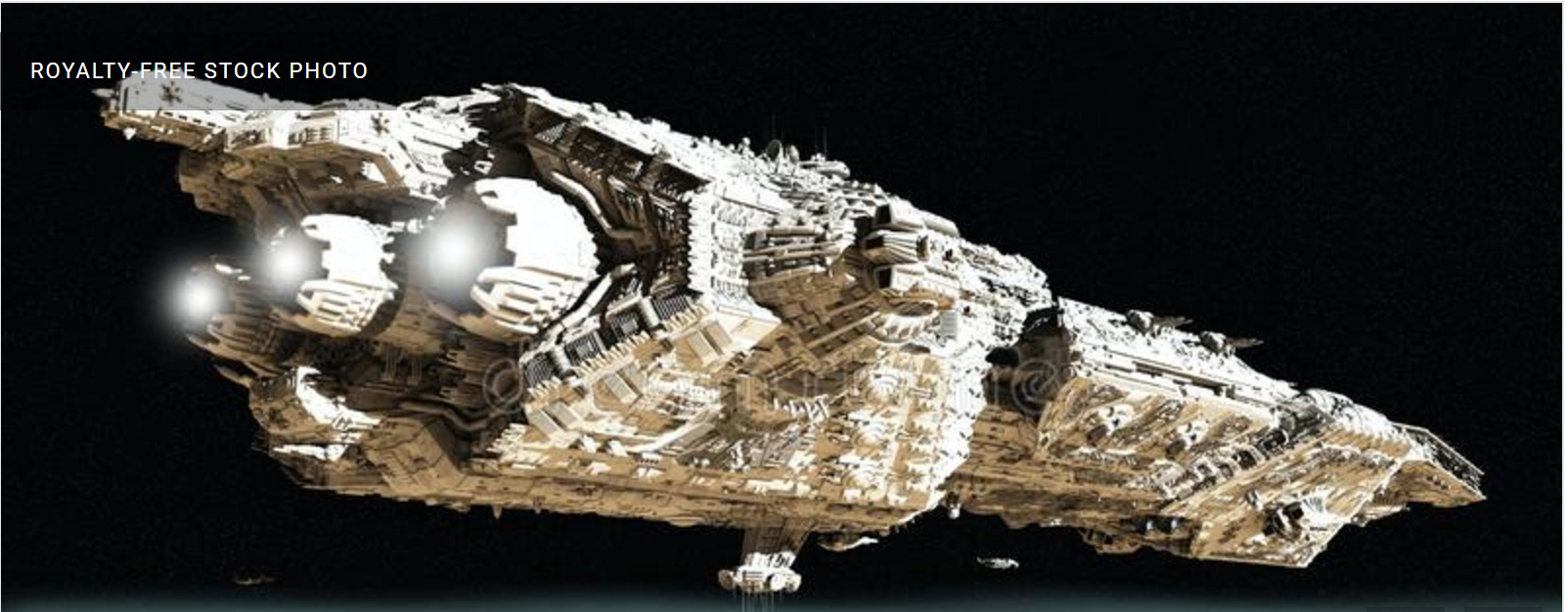


Image obtained from: https://www.dreamstime.com

|  |
| --- |
| CARRIER |
| name (String)  bayList  (an array of Fighter objects)  maxBays ( an integer)  numFighters ( an integer) |
| Constructor (takes the name of the Carrier and the maximum number of fighter bays)  loadFighter (Fighter\*)  launchNextFighter()  getInfo()  hasFighters()  getNumFighters()  getName() |

**State information:**

name - the name of the Carrier. It is a String e.g “Dreadnaught”

bayList  (an array of Fighter objects)

maxBays - an integer representing the maximum number of Fighters bays on the Carrier

numFighters - an integer representing the number of Fighters on the Carrier

**Behaviour information:**

Constructor (takes the name of the Carrier and the max number of Fighters it can hold)

loadFighter(Fighter\* ) takes a Fighter object and adds it to the array of Fighters if there is space.

**Fighters are added at the end of the array.**

**If a successful load occurs return true**

**Otherwise return false**

launchNextFighter() removes the next Fighter (at the head of the array)

**return the Fighter\* at the top of the list**

**If there were no Fighters on the Carrier return false**

printInfo() returns a string containing the following Carrier information:

**The Carrier name**

**The information of each Fighter on the Carrier**

hasFighters() returns a Boolean value indicating if the Carrier has Fighters available

getCapacity() returns the maximum capacity of the bus.

getName() returns the name of the Carrier

getNumFighters () returns the number of Fighters on the bus

**The FIGHTER class:**



Image obtained from: https://www.dreamstime.com

|  |
| --- |
| FIGHTER |
| fName (a String)  damage (an integer)  structStrength (an integer) |
| Constructor (takes the fighter name, the damage and structural strength)  reduceStructure(int damage)  geFName()  getStructStrength()  getInfo() |

reduceStructure(int damage) –reduces the structural strength by the parameter damage

getInfo()- returns a string containing the following Fighter information.

**The Fighter name**

**The structural strength of the fighter**

**The damage done by the fighter**

getFName()- returns the name of the Fighter

getDamage() - returns the damage done by the fighter

getStructStrength() - returns the structural strength of the Fighter

A text file “**shipData.txt**” contains all the information on each Carrier and all their Fighters.

The text file is organized as follows. The first line contains the name of the first Carrier. The next line contains two integers ***m*** and ***n***. The first number (***m***) represents the maximum number of bays on the Carrier and the second ***(n***) represents the actual number of Fighters on the Carrier.

Information on the next ***n*** fighters is then listed as follows. A single line contains a Fighter’s name. The subsequent line contains the Fighters structural strength followed by the damage it does (both in the same line). This is done for the each of the ***n*** Fighters.

The information for the second ship is in the same format of the first following the information for the last Fighter of the first Carrier separated by an empty line. Sample data is given below.

**Dreadnaught**

**25 3**

**Drifter**

**800 46**

**Skiff**

**1200 34**

**Adrono**

**700 38**

**Doomsday**

**15 2**

**Frigate**

**1300 67**

**Behemoth**

**2000 12**

**Battle Description**

A method called “battle” is responsible for simulating the battle between the two carriers.

The signature for the battle is as follows:

**void battle (Carrier c1\*, Carrier c2\*)**

A battle takes place as follows.

**Step 1:** Each Carrier launches one (1) Fighter. Let us call them f1 and f2 (The Fighters are removed from the Carrier arrays).

**Step 2:** the Fighters attack each other in a series of rounds until one is destroyed.

**Step 2.1:** At the beginning of each round either f1 or f2 attacks first. This **MUST** be chosen randomly every round.

**Step 2.2:** The attacking Fighter makes a random roll out of 100 to see if it successfully hits the other fighter. A roll of 50 and above indicates a “hit” and anything below 50 indicates a “miss”.

**Step 2.3:** In the case of a “hit”, a random roll from 1 to the damage of the attacking fighter is performed, and the attacked/defending fighter has its structural strength reduced by that amount.

If the fighter hits ,you must print “\*\*\*\*<f1 name> hits <f2 name> for <damage roll> \*\*\*\*\*”

If the Fighter misses print “\*\*\*\* <f2 name> misses \*\*\*\*”

**Step 2.4:** If the attacked/defending fighter is not destroyed, it counter attacks the first fighter and performs steps 2.2 to 2.3.

**Step 3:** The surviving fighter is loaded back onto its Carrier (using the loadFighter() method.

**Step 4:** repeat Steps 1 to 3 until one Carrier has no more ships.

**When either fighter is destroyed you must print “BOOOOOMMMM <fighter name destroyed!!!!!”**

**The Battle Simulator Class: (main class)**

**The main class is responsible for loading the Carriers from the text file shipData.txt and simulating the battle between the two Carriers by calling the function “battle” outline above.**

**The main class must be called “Battle\_Simulator”**

**Step 1:**

You main class should create and load both Carrier objects with data from the file **shipData.txt**

**Step 2:**

Your main class should then print the information for each Carrier

**Step 3:**

Your main class should simulate a battle between the two carriers by calling the function “battle” and passing both Carrier objects to it.

**Step 4:**

Your main class should then print the information on each Carrier (one Carrier should have no ships to print)

**Additional Info:**

Use srand((unsigned)time(0)); - as the random generator seed signature.

Use roll = rand()% number +1; - to generate a random integer from 1 to number

**You will need the following include statements at the top of you main**

#include<iostream>

#include<string>

#include<cstdlib>

#include<ctime>

#include<sstream>

#include<fstream>

**Below is an example of how to return a string using the ostringstream object from the sstream library**

string returnDetails(){

ostringstream ss;

ss<<"Hello"<<”world”<<"\n";

ss<<"More Text"<<variable<<"\n";

ss<<"Damage :"<<damage<<"\n"<<"\n";

return ss.str();

}

**All your objects (Carriers and Fighters) must return strings to be displayed from the main program. No printing should be done from the Carrier and Fighter objetcs.**

**Submission Requirements:**

**You and your partner’s Student ID number must be commented at the top of the file.**

**You must submit ONLY the C++ (cpp) file (Assignment1.cpp).**

**The marking scheme is given on the next page.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Trait** | **Excellent** | **Satisfactory** | **Unsatisfactory** |
| **Delivery**  **(4 marks)** | * Submitted on time and in the correct format. * Completed 90 - 100 percent of the program requirements | * Submitted on time and in the correct format. * Completed 70 -80 percent of the program requirements | * Submitted late or in the wrong format. * Completed less than 70% of the program requirements |
| **Coding Standards and Documentation (4 marks)** | * Includes name, date and assignment number. * Excellent variable names used (no global variables, or vague naming). * Useful documentation descriptions. * All functions commented. * Indented to standard. | * Includes name, date and assignment number. * Appropriate variable names used (a few use of global variables, or vague naming). * Basic documentation descriptions including purpose for functions. * Mostly indented well | * No name, program description included * Poor or misleading variable names used. Little or no indentation. * Regular use of global variables |
| **Specification and Runtime (8 marks)** | * The program meets all of the specifications required and works. * No errors in output. Output is formatted excellently. * All requirements met | * No errors in output. * Output has basic formatting or meets core specifications only. | * Does not run due to errors, data read incorrectly. * Little or no requirement met. * Output is poorly formatted or does not follow specifications. |
| **Efficiency**  **(4 marks)** | * Algorithm is easy to understand and efficient. * Can be maintained or modified with minimal changes | * Algorithm is easy to understand and but inefficient (excessive use of variables, loops or conditionals) | * Algorithm is hard to understand and very inefficient (excessive use of variables, loops or conditionals) |