| | |
|---|---|
| **Document Title** | **Lab 13** |
| **Course** | **CS-200** |

**MAXIMUM TIME ALLOWED 3: Hours**

- Always keep a notebook and a pen with for any programming assignment
- First understand the problem, plan how you will tackle it on the notebook
- Write a rough pseudo code for your solution to the problem and then start to code.

It will be a lot easier and time saving; otherwise you will hit a wall right in the middle of writing code because you hadn't thought about the problem fully.

**You can take help from lecture slides uploaded on LMS.**

Also you will be marked for

- proper variable naming (10 mark deduction if this is not done)
- proper indentation (10 mark deduction if this is not done)

Indentation helps make the code readable and easy to understand. A code which is not indented is just like a text with no full stops, commas, paragraphs or general punctuation whatsoever, it is almost impossible to read.

Code has to be indented after every curly bracket '**{**'. And whenever those brackets are closed '**}**', the remaining code is not indented. Look at the following.

```
int main(){
        // I will write my code here because of a curly bracket
        // ok let me open another curly bracket
        {
                // now I am indenting more than before
        }
        // back again
}
```

Use tabs for indenting your code so that it is readable. Open a cpp file by writing -

```
nano –i task1.cpp
```

This allows Nano to do some indenting on its own. But you have to indent each first line after a curly bracket by yourself; it then remembers the level of indentation for the next lines. Perhaps this may be more useful when you write code with many levels of indentation!

**All grading will be done in lab so finish your lab work in time! No grading after the lab is over and upload your work.**

**You can take help from slides and the book**

## Task1 [10 points]:

To begin with create a class called Tool with a single integer variable called 'strength' and a string variable called 'type' which will store the name of the tool. In the Tool class, create a function setStrength which takes an integer and sets the strength to that integer. Lastly, to make this an abstract class, define a pure virtual function fight like this:

***virtual bool fight(Tool *t) = 0;***

It is important to pass the Tool object using a pointer which invoked a process known as 'dynamic binding' which is used when the exact type of an object is unknown.

Note that this function has no body and it has been a set a value zero. This tells the compiler that the function is a virtual function and that its definition has not been provided on purpose (because we will provide it in the derived classes).

## Task2 [20 points]:

Create three more classes called Rock, Paper and Scissors which inherit from Tool. Create constructors which take a single integer argument for the classes in which you define the strength of each tool (remember that all objects of derived classes are also objects of base classes) using the ***setStrength*** function. Also, define the type of the tool, 'rock' for Rock, 'paper' for Paper and 'scissor' for Scissor.

## Task3 [20 points]:

Now that we have our base code up, we can finally start to implement the ***fight*** function in our derived classes to complete our functionality and allow us to play the game. In our declarations of the ***fight*** function we do not need to type virtual again although typing it will not cause an error. Just the normal declarations will work. This function will take in a Tool object as argument. So the prototype will be:

***bool fight(Tool *t);***

The actual definition for the function will be different for all three classes allowing us to call just one function on different objects and it will behave according to the object it was called on.

To implement this, when rock fights scissors, its strength is temporarily doubled and thus it is able to beat scissors. However, when it fights paper, its strength is reduced to half allowing paper to beat it. Extend this for all three classes. The function should return true if the tool beats its competitor and false otherwise.

---

## Exception handling

## Task4 [20 points]:

You'll need to modify the IntArray class by Adding a SubscriptError exception class definition inside the IntArray class of "IntArray.h". Remove the subscriptError function in

"IntArray.cpp". Replace all calls to the subscriptError function by throwing the SubscriptError exception. Modify the main program in main.cpp to utilize the try-catch

four times -- once for each IntArray usage of its operator[]. In each catch part print a meaningful error message.

## Task5 [30 points]:

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

int main() {
   int num = 0;

   do {
      cout << "Enter a number between 1 and 10" << endl;
      //cin.exceptions(iostream::failbit);
            cin >> num;

      if(num < 1 || num > 10) {
         cout << "Illegal value, " << num << " entered. Please try again." << endl;
      }
   } while (num < 1 || num > 10);

   cout << "Value " << num << " correctly entered! Thank you." << endl;
}
```

The program is requesting a number, but what if the user does not cooperate? Run the program again and enter 'four'. The program is unable to store an integer value into num, so num retains its original value of 0 and the program falls into an infinite loop. By default, ***cin class*** does not throw exceptions when something goes wrong with IO. Activate exception-throwing by uncommenting this line of code in your main method.
cin.exceptions(iostream::failbit);

This line informs ***cin class*** that you would like the program to throw an *iostream::failure* exception whenever an io-related method fails. Try running the program again and enter "four". This time, the program should terminate and generate an error message, rather than fall into an infinite loop.

Now it's time to handle the exception we've activated. Add a try/catch block to catch and handle the *iostream::failure* exception. Identify the statements that cause the error as well as the portions of the program that depend upon these statements. Enclose these statements within the try block. Follow the try block with the catch block given below. Note that we pass the exception by reference. Also note that, when cin throws its exception, the input token will remain in the buffer so that it can be examined by the program. In our case, we will not be examining the token, but simply clearing out the buffer to start over.

```
catch(iostream::failure& iof) {
    cout << "Non-integer value. Please enter a number." << endl;
    cin.clear();                          // reset error flags
 cin.ignore( std::numeric_limits<int>::max(), '\n');    // clear buffer
}
```

Try running the program again and entering 'four'. If you have placed your try/catch block appropriately, the program will inform you this is an invalid value, and then it will give you the opportunity to try again. Now our program catches one type of invalid input, but is there more? Recall that the program is requesting a number between 1 and 10. Run the program again and enter 10.8. Although this number is greater than 10, the program will think you entered simply 10 and tell you it is a valid value. In fact, the program will accept any invalid input so long as it *starts* with an integer, leaving anything else for the next time some input is requested from the buffer. ***cin class*** does not offer an exception for this situation, so we'll have to make our own. Use cin.peek() function call to get the char after the integer it put in ***num***. If it is not equal to '\n' then we should throw our own exception of type NotAValidNumber. For this you will have to make a new class of this type.

---

**Format of name in Lab submissions**

**<Student-ID>-Lab<#>-<TA-ID>.zip**

For example 18100075-Lab4-Huzaifa.zip means this is the lab submission of 18100075 for Lab4 checked by Huzaifa. This way it will be easier for students and TAs to figure out the TA in case of any problems.