CEng 242 Homework 5

Due: 12^{th} May 2000

In this homework, you will write a class library for a very simple strategy game. In this game you have players which may have ability to fight and/or cast spells. They start with an initial strength and mana (ability to cast spell) power and they can collect some tools to increase their powers. They have a standart attacking strategy. And a standart one-on-one fight procedure is defined between two players.

In the implemantation you should have a base class `Player` with the following definition:

```
#define MAXATTACK 4
#define MAXSPELL 4

struct Move {
   unsigned hit;
   unsigned cast;
};

class Player {
protected:
   char name[40];
public:
   Player(char *s) { strncpy(name,s,40);name[39]='\0';}
   virtual ~Player () { cout << name << " is death!\n";}
   const char *getname() { return name;}
   virtual unsigned strength()=0;
   virtual unsigned mana()=0;
   virtual void operator++() {};
   virtual void operator--() {};
   virtual unsigned loosestrength(unsigned i) {return i;};
   virtual unsigned loosemana(unsigned i) {return i;};
   Move attack();
   void operator*(Player &);
   friend ostream & operator<< (ostream &,Player &);
};
```

`strength()`
: A virtual function which returns the positive integer representing fighting power of a player, should be implemented on derived classes.

`mana()`
: A virtual function which returns the positive integer representing power of casting spells of a player, should be implemented on derived classes.

++ operator
: A virtual function representing that the player physically strengthen because of a found food or moral event like killing some enemy. Has a null implementation as the default.

`--` operator
: A virtual function representing that the players mana is increased because of a newly found spell script or a moral event. Has a null implementation as the default.

`loosestrength(unsigned i)` A virtual function to decrease players strength in the fight. It returns the remaining amount if player looses all of his `strength()`. It should be implemented in the derived class if player is a fighter having a positive `strength()`. If `strength()` is less than `i` it becomes `0` and `i-strength()` is returned. If `strength()` is greater than or equal to `i`, it is decremented by `i` and and `0` is returned.

`loosemana(unsigned i)` A virtual function to decrease players mana. It returns the remaining amount if player looses all of his `mana()`. It should be implemented in the derived class if player is a wizard having a positive `mana()`. If `mana()` is less than `i` it becomes `0`. If `mana()` is greater than or equal to `i`, it is decremented by `i` and and `0` is returned.

`attack()`
: This function should be implemented in the player class but not in derived classes. It returns the attacking move of the player. A `Move` structure consists of two components. `hit` is the physical attacking move of the player and decreases the players `strength()`. `cast` is the spell casting move of the player and decreases players `mana()`. `hit` is defined as the minimum of `MAXATTACK` and `strength()/2`. `cast` is defined as the minimum of `MAXSPELL` and `mana()/2`. Function decreases the `strength()` and `mana()` values of the players by `(hit+1)*2/3` and `(cast+1)*2/3` respectively and return the `Move` structure representing the attack.

`operator*(Player &enemy)`
: This operator simulates the fight between the object and the player `enemy`. First the object starts attacking. Attacking turn changes in a loop until no parts has attacking powers (`attack()` function returns `(0,0)`) or any of the parts is death. In case of the death you should raise (`throw`) the pointer of the died object as an exception. Each turn can be described as follows:

1. Get the `attack()` value from the attacker.
2. Loose the `hit` value from the defenders strenght, save the remaining hit value.
3. Loose the `cast` value from the defenders mana, save the remaining cast value.
4. Loose the remaining `hit` value from the defenders *mana* by $(x + 1)/2$ for hit value x.
5. Loose the remaining `cast` value from the defenders *strength* by x for cast value x.
6. Check if the defender has one of his `strenght()` or `mana()` values are positive or not. If defender is death, apply ++ and `--` as a moral support to attacker and throw the pointer of the defender object as an exception indicating the death.
7. Change the roles of the players for next turn.
8. Loop to 1. until both players has an attack power of 0 `hit` and 0 `cast`.

`ostream &operator<<(ostream &,Player &)`
: Overloaded operator << displays the player information on a stream with the following format:

```
Name: name
Strength: strength() Mana: mana()
```

You are not allowed to change the definition of class `Player`. You should only implement the unimplemented functions. Then you should derive the following classes directly or indirectly from players.

Peasant
Derived from `Player`. Peasants have an `unsigned` attribute `power` to store the `strength()`. Since peasants cannot cast, they always return 0 for `mana()` requests. Initial value of `power` is 1 and for all ++ requests it is incremented by 1. Their `power` cannot be greater than 10. All extra increments will be ignored.

Warior
Derived from `Player`. Wariors have an `unsigned` attribute `power` to store the `strength()`. Similar to peasants, they can not cast spells. However their initial `power` is 5, there is no limitation on their strengths and ++ requests increments their strength by 2.

Wizard
Derived from `Player`. Wizards have no fighting capabilities but they use spells to attack their enemies or defense themselves. They have a `unsigned` attribute called `spell` to store their `mana()`. Their initial `spell` is 5 and -- request increments their `mana()` by 2. There is no limitation on their `mana()` but they always return 0 for `strength()`.

Darklord
Derived from both `Wizard` and `Warior`. They are very specially trained wariors who have also a good knowledge of spells. They have initially 5 as `power` and `spell`. In the definition of the `Darklord` class, you only need the definition of the constructor, `strength()` and `mana()`. Any other attribute or declaration is neither necessary nor allowed in the definition of `Darklord`.

All players should have the constructors implemented with a single `char *` parameter which stands for the name of the player and should be passed to the constructor of `Player` class. All member definitions in `Player` class should be completed-implemented so they can be usable from any player instance.

Also you will define an `Army` structure with the following definition:

```
struct Army {
   char name[40];
   Player *soldiers[100];
   int n;

   Army(char *s) { strncpy(name,s,40);name[39]='\0';n=0;}
   ~Army();
   void operator+=(Player *p) { soldiers[n++]=p;}
   void operator-=(unsigned);
   void operator-=(Player *);
   Player & operator[](int i) { return(*soldiers[i]); }
};
```

name is the name of the army. soldiers keeps the pointers of the army members. Army contains n soldiers. So first n element of the soldiers array is used. += request is used to add a new Player to the army. For example:

```
Army a("Kifirbizan");
a += new Warior("Hosterbulgan");
a += new Darklord("Hinzurgabon");
a += new Peasant("Paspason");
a += new Wizard("Hokustonkapus");
```

Will define the army Kifirbizan consisting of 4 soldiers with all different types. You have to overload the -= i operator for an unsigned integer to delete the $\mathtt{i}^{th}$ soldier from the soldiers array and -= p for a Player pointer to match and delete the player with pointer p (you can call the integer version to delete). In deletion you should shift the pointers up to fill the deleted position. Note that used elements of the array soldiers are always in the range $[0..n-1]$. Also you should deallocate the death players memory. Also overload the operator ostream &operator<<(ostream &,Army &) to list all soldiers in the army in the following format:

```
Army: Kifirbizan
  Hosterbulgan 5 0
  Hinzurgabon 5 5
  Paspason 1 0
  Hokustonkapus 0 5
```

When your implementation is finished, the following *war* procedure should work for two armies A and B:

```
int ia=0,ib=0,count=0;

while (count<1000000L && A.n && B.n) {   //  While no army is defated
  cout << A[ia] << B[ib];
  try { A[ia]*B[ib]; } //  Two soldiers fight
  catch (Player *p) {  //  p is death
    if (p==A.soldiers[ia]) { //  if soldier of A
       A-=ia; //  delete the death soldier from army A
       ia--; //  correct ia since soldiers are shifted
    } else {
       B-=ib;  //  delete the death soldier from army B
       ib--;   //  correct ib since soldiers are shifted
    }
  }
  ia++; if (ia>=A.n) ia=0;
  ib++; if (ib>=B.n) ib=0;
  count++; //  to avoid infinite loop of two powerless soldiers
}
```

Submission details will be announced later.