OOP with examples

Introduction

First of all, why? The object-oriented ideology was developed as an attempt to associate the behavior of an entity with its data and to project objects of the real world into program code. It was assumed that such code is easier to read and understand by a human.

It is known that a person is more accustomed to perceiving the world around him as a set of classified objects interacting with each other.

Classes and Objects

Class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. The variables inside class definition are called **data members** (**fields**) or **properties** and the functions are called **member functions** (**methods**) or **behavior**.

Note: The only difference between an object method and a regular function is that the method has access to its own state via fields.

> For example, Class of birds, all birds can fly and they all have wings and beaks. So here flying is behavior and wings and beaks are part of their characteristics. And there are many different birds in this class with different names but they all possess this behavior and characteristics. Similarly, the class is just a blueprint, which declares and defines characteristics and behavior, namely data members and member functions respectively. And all objects of this class will share these characteristics and behavior.

Objects are instances of a class, which holds the data variables declared in class and the member functions work on these class objects. Each object has different data variables.

Properties and methods form the **interface of an object (class)**, and entities with a particular combination of properties are **instances of the object (object)**. Let us give a definition of OOP based on the introduced concepts.

Object-oriented programming is a methodology for developing programs as a collection of objects belonging to certain classes and interacting through each other's properties and methods to perform operations and transform data, determining the behavior of their instances.

Some rules for creating classes:

— The class name must start with an uppercase letter (but this is not mandatory). If the class name is made of more than one word, then the first letter of each word must be in uppercase. Example, class Study, class StudyTonight, etc

— Classes contain, data members and member functions, and the access of these data members and variables depends on the access specifiers.
— Class's member functions can be defined inside the class definition or outside the class definition.
— Class in C++ are similar to structures in C, the only difference being, class defaults to private access control, whereas structure defaults to public.
— All the features of OOPS, revolve around classes in C++: Inheritance, Encapsulation, Abstraction, etc.
— Objects of class hold separate copies of data members. We can create as many objects of a class as we need.
— Classes do posses more characteristics like we can create abstract classes, immutable classes.

Creating a class interface

To define such a data type as a class in the C++ language, the **class** keyword is used. The use of the class keyword defines a new user-defined data type. For example:

```
class Character {
}
```

Now you can add fields and properties to the class. The field is declared like a regular variable:

```
class Character {
    int health;
    int attack;
    int armor;
};
```

Everything inside the curly braces belongs to this class. Even if the class is still empty, you can already create an instance of it - an object. This is called declaration or instantiation.

```
Character hero = new Character();

Character person;
```

This is similar to how variables are created, but the name of the class is specified instead of the data type. The assignment sign is followed by the **new** keyword, which must be used to create a new instance of a class, and a constructor, a special method that allows you to create an object.

You already know that defining a variable of a fundamental data type (eg int x) results in memory being allocated for that variable, just as creating an object of a class results in memory being allocated for that object.

TASK FOR INDEPENDENT SOLUTION:

1. Add fields **name** (string), **level** (integer from 0 to 100) to class Character.
2. Create an **class Name** with two fields: 1) first name, 2) second name.
3. Replace the string Character Name with an object of the class Name.

But if we start working with this object as a structure in C, we get an **error**:

```
class Character {
    int health;
    int attack;
    int armor;
};

int main () {
    Character person;
    person.health = 100;   // error - field not available
    person.attack = 100;   // error - field not available
    person.armor = 100;    // error - field not available
}
```

SOLUTION 1

Now the object has its own fields, but they cannot be accessed from the outside, because access is denied (more on this in the section on encapsulation). To open it you need to use the **public** keyword.

```
class Character {
  public:
    int health;
    int attack;
    int armor;
}

int main () {
    Character person;
    person.health = 100;   // work
    person.attack = 100;
    person.armor = 100;
}
```

If access to the fields is open, then you can perform calculations with them or simply get their value. If you need to deny access to certain fields, use the **private** access level, then they will be available only from inside the class.

SOLUTION 2

Inside the hidden properties of an object, you can often find two constructions (more on this in the section on encapsulation):

1. get (**getter** - allows you to get the property value). It returns the value that comes after the **return** keyword;
2. set (**setter** - allows you to change the value). It specifies the field to be changed using the value *value*.

```
class Character {
    int health;
    int attack;
    int armor;
  public:
    int getHealth() { return this.health; }
    void setHealth(int _healthPoints) { health = _healthPoints; }

    int getAttack();
    void setHealth(int _attackPoints);

    int getArmor();
    void setArmor(int _armorPoints);
}
```

Such manipulations are necessary so that access to the fields is carried out only as the developer needs it.

TASK FOR INDEPENDENT SOLUTION:

1. Add ALL POSSIBLE getters and setters to your Name class and Character class.

Static fields and class methods

The getHealth() method, despite being common to all class instances, **applies to a specific object and operates on its specific field**. In addition to such class fields and methods, you can also define static fields and methods. **Static members** of a class belong to the class, not to its instances. That is, they will be common to all created objects.

For example, let's add a static count field to our Character class, which will increase by 1 every time a new instance is created, that is, we will count the number of characters that already exist. A static field is created using the **static** keyword.

```
class Character {
    int health;
    int attack;
    int armor;
  protected:
    static size_t count;
  public:
    ...
}
```

Features of static class fields:

1. A static field is accessed through the class name

```
Character::count
```

2. However, you can also access a static field through the this pointer

```
this->count
```

3. The static field needs to be initialized. Initialization occurs anywhere in the cpp file with the class implementation. In our case:

```
size_t Character::count = 0;
```

4. A static variable is declared as **protected**.

Now we need to somehow access our variable. To do this, we will write a method that returns the count value.

```
class Character {
    int health;
    int attack;
    int armor;
  protected:
    static size_t count;
  public:
    static size_t getCount();
    ...
}

size_t Character::count = 0;

size_t Character::getCount() {
    return Character::count;
}
```

Since the method accesses a static field, it does not need an instance of the class in order to be called. Therefore, the getCount method is made static.

You can now access the method through the class name Character::getCount() However, you can access the static method through the class instance (hero.getCount()), but this is **bad practice**.

If the method is static, then it can be accessed before at least one instance of the variable has been created. Therefore, a static method cannot access non-static class fields. In addition, a static method simply will not know which instance to refer to - inside it there is no access to the this pointer.

Static fields and methods, as well as non-static ones, can have private, protected, and public access modifiers. When using static methods, the const method modifier is prohibited, since the method, by definition, cannot access and change non-static fields.

TASK FOR INDEPENDENT SOLUTION:

1. Add a static field numberOfCalls to store the number of calls to the fields of objects of the Character class. Each time, when a setter or getter is called, this field is incremented by +1. Add the getCallsCount() method.

Basic concepts for writing simple classes

1. **this** — is a special local variable (inside class methods!) that allows an object to access its own attributes from its methods.

Example: outside (for example, in an application using this class), the health field access will look like this:

```
hero.health
```

If the object wants to refer to its x field itself, in its method the call will be this.health (as in the getHealth() method of the example above) or simply health (as in the setHealth(int) method of the example above). That is, using the this word is optional, but it makes the code more readable.

2. **constructor** — it is a special method that is automatically called when an object is created.

Objects are initialized using special class functions called **Constructors**. And whenever the object is out of its scope, another special class member function called **Destructor** is called, to release the memory reserved by the object. C++ doesn't have Automatic Garbage Collector like in JAVA, in C++ Destructor performs this task.

A constructor can take any arguments just like any other method. In each language, the constructor is denoted by its own name. In C++, the constructor name must be the same as the class name. The purpose of the constructors is to perform the initial initialization of the object, fill in the required fields, and possibly allocate the required memory.

The main types of constructors:

— default constructor;

— initialization constructor;

— copy constructor.

There are other kinds of constructors, for example, type conversion constructor, move constructor.

Multiple constructors with the same name but different parameters are allowed.

```
class Character {
    int health;
    int attack;
    int armor;
  public:
    Character();                    // default constructor
    Character(int, int, int);       // initialization constructor
    Character(const Character&);    // copy constructor
    ...
}
```

```
Character::Character() {
  health = 100;
  attack = 100;
  armor = 100;
}

Character::Character(int _health, int _attack, int _armor) {
  health = _health;
  attack = _attack;
  armor = _armor;
}

Character::Character(const Character& hero) {
  health = hero.health;
  attack = hero.attack;
  armor = hero.armor;
}
```

TASK FOR INDEPENDENT SOLUTION:

1. Add 3 primary constructors (default constructor, initialization constructor, copy constructor) to the Name class.

If the class has fields that require memory allocation, this memory is allocated in the class constructors. Consider the same example with the Character class, which has two more fields: the number of character skills and an array of these skills. In the constructors of this class, code sections appear with memory allocation for skills.

```
class Character {
    int health;
    int attack;
    int armor;
    int numOfSkills;
    string* skills;
  public:
    Character();                      // default constructor
    Character(int, int, int, int*);   // initialization constructor
    Character(const Character&);      // copy constructor
    ...
}

Character::Character() {
  health = 100;
  attack = 100;
  armor = 100;
  numOfSkills = 3;
  skills = new string [3];
  skills[0] = "high intelligence";
  skills[1] = "magic";
  skills[2] = "craft";
}

Character::Character(int _health, int _attack, int _armor,
  int _numOfSkills , int* _skills) {
  health = _health;
  attack = _attack;
  armor = _armor;
  numOfSkills = _numOfSkills;
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = _skills[i];
  }
}

Character::Character(const Character& hero) {
  health = hero.health;
  attack = hero.attack;
  armor = hero.armor;
  numOfSkills = hero.numOfSkills;
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = hero.skills[i];
  }
}
```

TASK FOR INDEPENDENT SOLUTION:

1. Add the fields achievements (array of strings) and achievementsCount (integer) to the Character class. Fix the constructors by adding initialization of the added fields.

IMPORTANT NOTE!

Some data types (for example, constants and references) must be initialized right away, for example,

```
class Character {
    int health;
    int attack;
    int armor;
    int numOfSkills;
    string* skills;

    const int bonusToMagicAttack;

public:
   Character() {
       health = 100;
       attack = 100;
       armor = 100;
       numOfSkills = 3;
       skills = new string [3];
       skills[0] = "high intelligence";
       skills[1] = "magic";
       skills[2] = "craft";

       bonusToMagicAttack = 20;  // ERROR: constants are not allowed
                                 // assign values
    }
    ...
};
```

SOLUTION

To solve this problem, C++ added a method to initialize class member variables through a member **initialization list**, instead of assigning values to them after the declaration.

> Note. Generally speaking, variables can be initialized in three ways: through copy initialization, direct initialization, uniform initialization.
>
> ```
> int value1 = 3;         // copy initialization
> double value2(4.5);     // direct initialization
> char value3 {'d'}       // uniform initialization
> ```

Using an initialization list is almost identical to doing direct initialization. An example of writing an initialization constructor using an initialization list:

```
Character::Character(int _health, int _attack, int _armor, int _numOfSkills ,
int* _skills) : health(_health), attack(_attack), armor(_armor),
numOfSkills(_numOfSkills), bonusToMagicAttack(20) {
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = _skills[i];
  }
}
```

The member initialization list is located immediately after the constructor parameters. It starts with a colon (:) and then the value for each variable is given in parentheses. You no longer need to perform assignment operations in the body of a constructor. Also note that the member initialization list does not end with a semicolon.

The above code works because we are allowed to initialize const variables (but not assign values to them after they are declared!).

Constructor methods operate on internal state and are otherwise the same as regular functions. Even the declaration syntax is the same.

TASK FOR INDEPENDENT SOLUTION:

1. Use initialization lists to implement your 3 primary constructors (default constructor, initialization constructor, copy constructor) in Name class.

Let's look at the example when our class had a static field count, which counts the number of instances of the class. If this field has a place to be, then in the implementation of constructors, you need to increase the value of this counter, for example,

```
Character::Character(const Character& hero) {
  health = hero.health;
  attack = hero.attack;
  armor = hero.armor;
  numOfSkills = hero.numOfSkills;
  skills = new string [numOfSkills];
  for (int i = 0; i < numOfSkills; ++i) {
     skills[i] = hero.skills[i];
  }
  Character::count++;
}
```

3. **destructor** — it is a special type of class method that is executed when an object of the class is deleted. While constructors are meant to initialize a class, destructors are meant to clean up memory after it.

When an object is automatically out of scope, or a dynamically allocated object is explicitly deleted using the **delete** keyword, the class destructor (if it exists) is called to perform the necessary cleanup before the object is removed from memory.

For simple classes (those that only initialize the values of ordinary member variables), a destructor is not needed, since C++ will automatically clean up itself. If the class object contains any resources (e.g. dynamically allocated memory or a file/database), or if you need to do something before the object is destroyed, the destructor is ideal because it is the last thing that happens to object before its final destruction.

```
class Character {
    int health;
    int attack;
    int armor;
    int numOfSkills;
    string* skills;
  public:
    Character();                          // default constructor
    Character(int, int, int, int*);       // initialization constructor
    Character(const Character&);          // copy constructor

    ~Character();                         // destructor
    ...
}

Character::~Character() {
  // dynamically delete the array that was allocated in the constructor
earlier
  delete[] skills;
}
```

Comment. If you have a static count field, don't forget to decrement the counter value by 1 inside the destructor.

TASK FOR INDEPENDENT SOLUTION:

1. Edit the destructor with the achievement field you added earlier.

2. Add processing of field count. See comment before this task.

A NOTE on the topic "WHY DO WE NEED CONSTRUCTORS AND DESTRUCTORS?"

By using constructors and destructors, your classes can initialize and clean up after themselves automatically without you having to do anything! This reduces the chance of errors and simplifies the process of using the classes.

4. Namespace — this is the namespace in which the class resides. It is necessary so that there are no conflicts with the names of classes and variables from the included libraries. The namespace is specified in the form: **namespace name (class) ::**

Basic principles of OOP

All object-based languages (C#, Java, C++, Smalltalk, Visual Basic, etc.) must adhere to the three basic principles of OOP, which are listed below:

1. **Encapsulation** is a property of the system that allows you to combine data and methods that work with them in a class and hide implementation details from the user.
2. **Inheritance** is a property of the system that allows you to describe a new class based on an existing one with partially/fully borrowed functionality. The class from which you inherit is called the base, parent, or superclass. The new class is a child, inheritor, or derived class.
3. **Polymorphism** is a property of the system to use objects with the same interface without information about the type and internal structure of the object.

There is often an opinion that in OOP it is worth highlighting another important characteristic - abstraction. Officially, it was not included in the mandatory features of the OOP, but it should not be written off. Note that abstraction is inherent in any programming, not just OOP.

4. ? Abstraction is a way to isolate a set of significant characteristics of an object, excluding non-significant ones from consideration. Accordingly, an abstraction is a set of all such characteristics.

TASK FOR INDEPENDENT SOLUTION:

1. Consider the class system that has already formed and determine which principles and in what places are already reflected in your code.

Let's look at each principle in detail.

Basic principles of OOP: abstraction

Abstraction is a concept according to which characteristics are distinguished from an object that distinguish it from all other objects, clearly defining its conceptual boundaries.

The idea is to separate the way compound objects are used in a program from the details of their concrete implementation as simple objects. Abstraction reflects such a feature of the subject area as the identity and similarity of objects, and involves the selection of only those features and behaviors that are essential from the point of view of their software implementation.

This concept instructs programmers to separate the software implementation from the real object using the properties and methods of the class, declaring only what is essential from the point of view of the software implementation.

The ability to balance between the simplicity of the architecture and the flexibility of the application is an art. Choosing the golden mean, you should rely not only on your own experience and intuition, but also on the context of the current project.

The wrong choice of abstraction level leads to one of two problems:

— if abstraction is not enough, further extensions of the project will run into architectural restrictions, which lead either to refactoring and changing the architecture, or to an abundance of crutches (both options usually entail pain and financial losses),

— if the level of abstraction is too high, this will lead to overengineering[1] in the form of an overly complex architecture that is difficult to maintain, and excessive flexibility that will never be useful in this project. In this situation, any simple changes to the project will be accompanied by additional work to meet the requirements of the architecture (this also sometimes brings some pain and financial losses).

Abstraction levels (on the example of Character and game application)

First layer. The game has one character class, all properties and behavior are described in it. This is a very wooden level of abstraction, suitable for a casual game that does not involve any particular flexibility.

[1] The word "overengineering" is still more often used by computer scientists to denote unnecessarily cumbersome ("Hindu") code. Experts say that once Indian programmers were paid for the lines of programs they wrote, so they gave out complex, voluminous, ornate code. Overengineering in the production sense has two aspects: manufacturing a product in a way that is too complicated and costly - "from a cannon to sparrows", and the product itself can be overengineered - with a higher quality than the client requires, with unnecessary features and options.

Second level. The game has a basic game character with basic abilities and character classes with their own specialization (such as mage, warrior, shaman), which is described by additional properties and methods. The player is given a choice, and it is easier for developers to add new classes.

Third level. In addition to the classification of characters, aggregation is introduced using a system of slots (for example, weapons, scrolls, potions). Some of the behavior will be determined by what the player has set in their character. This gives the player even more options for customizing the game mechanics of the character, and gives developers the opportunity to add these same expansion modules, which in turn makes it easier for game designers to release new content.

Fourth level. Components can also include their own aggregation, which provides the ability to select materials and parts from which these components are assembled. This approach will give the player the opportunity not only to stuff the characters with the necessary components, but also to independently produce these components from various parts. This level of abstraction in games is rare, and for good reason. This is accompanied by a significant complication of the architecture, and balance adjustment in such games turns into hell.

An example of a problem for building abstractions

Abstraction 1 layer

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;
}
```

Abstraction 2 layer

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;
}

class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
}

class Warrior : public Character {
    int weaponDurability;
    int weaponAttack;
    int shieldPoints;
}
```

Abstraction 3 layer

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;

    Backpack objects;             // backpack object slots
}

class Backpack {
    size_t count;                 // number of objects in the backpack
    const size_t capacity;        // backpack capacity (by weight)

    Object* content;              // contents of the backpack
}

class Object {                    // game object
    string name;
    const size_t weight;
}

class Potion : public Object {
    string type;
    int pointsBonus;              // bonus points added by the potion
    int timeOfAction;
}

class Weapon: public Object {
    int weaponDurability;
    int weaponAttack;
}
```

```
class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
}

class Warrior : public Character {
    int weaponDurabilityBonus;
    int weaponAttackBonus;
    int shieldPoints;
}
```

TASK FOR INDEPENDENT SOLUTION:

1. Bring your workpiece to the 3rd level of abstraction. Select **10 or more** interacting classes. Start from your desire to implement some functions for this Game. YET ONLY NAMES AND CLASS FIELDS!
2. Add setters and getters for the added classes.
3. Add constructors for the added classes.
4. Add destructors as needed.

Hint: This abstraction example uses inheritance. It will be useful to read the topic of inheritance first, and then return to the implementation of points 2-4.

Basic principles of OOP: polymorphism

Polymorphism is a system property that allows you to have multiple implementations of a single interface. Let's look at an example.

Suppose we have three characters: a warrior, a magician and a shaman. The characters in our game are fighting, so they have the attack() method. The player, by pressing the “fight” button on his joystick, tells the game to call the attack() method on the character the player is playing as. But since the characters are different, and the game is interesting, each of them will attack in some way. For example, Oleg is an object of the shaman class, and shamans can call totems to attack.

The benefit of polymorphism in this example is that the game code does not know anything about the implementation of its request, who should attack how, its task is simply to call the attack() method, the signature of which is the same for all character classes. This allows you to add new character classes, or change the methods of existing ones, without changing the game code. It's comfortable.

Basic principles of OOP: inheritance

Inheritance is a system mechanism that allows some classes to inherit the properties and behavior of other classes for further extension or modification.

What if we don't want to play for the same characters, but want to make a common image, but with different content? OOP allows us to do this trick by separating the logic into similarities and differences, and then pushing the similarities into the parent class and the differences into descendant classes.

So, in our example, the differences between shamans and warriors are only that shamans attack and defend with totems, while warriors attack with weapons and have a shield. All other properties and behavior will not make any difference. In this case, you can design the inheritance system as follows: common features (health, attack, armor) will be described in the base class "Character", and the differences in the two child classes "Shaman" and "Warrior".

If you override an already existing method in the parent class in the descendant class, then the overload will work. This allows not to supplement the behavior of the parent class, but to modify it. At the time of calling a method or accessing a field of an object, the attribute is searched from the child to the root itself - the parent. That is, if the Attack () method is called on the shaman, the method is first searched in the descendant class - Shaman, and if it is not there, the search goes one step higher - to the Character class.

It is curious that an overly deep inheritance hierarchy can lead to the opposite effect - complication when trying to figure out who inherits from whom, and which method is called in which case. In addition, not all architectural requirements can be implemented using inheritance. Therefore, inheritance should be applied without fanaticism.

There are guidelines to prefer composition over inheritance where appropriate.

Composition - the inclusion of a content object by a container object and control of its behavior; the latter cannot exist outside the former.

When describing the relationship of two entities, how to determine when inheritance is appropriate, and when composition is appropriate? You can use the popular cheat sheet: ask yourself, entity A is entity B? If yes, then most likely, inheritance is suitable here. If entity A is part of entity B, then our choice is composition. In relation to our situation, it will sound like this:

Is the Shaman a character? Yes, we choose inheritance. Is luggage part of the character? Yes, it means composition.

This cheat sheet helps in most cases, but there are other factors to consider when choosing between composition and inheritance. In addition, these methods can be combined to solve different types of problems.

Another important difference between inheritance and composition is that inheritance is static in nature and establishes class relationships only at the interpretation / compilation stage. Composition, on the other hand, allows you to change the relationship of entities at the moment.

TASK FOR INDEPENDENT SOLUTION:

1. In our game, there will be a set of special days (holidays) when legendary items will drop out for free for users. For this functionality, you will need the **Date** class (year, month, day) and the **Event** class inherited from it (+ the name, how many days it lasts). Implement the given class system (without constructors and methods yet).

Multiple Inheritance[2]

We considered the situation when two classes are inherited from a common child. But in some languages, you can do the opposite - inherit one class from two or more parents, combining their properties and behavior. The ability to inherit from multiple classes instead of just one is multiple inheritance.

Additional terms to be aware of

Delegation - assignment of a task from an external object to an internal one; **Aggregation** - inclusion by the container object of a link to the content object; when the former is destroyed, the latter continues to exist. It is recommended to read about these concepts on your own.

Note. Methods and properties that can be overridden are called virtual. In the parent class, the virtual modifier is specified for them. More on inheritance and method overriding will be discussed later. Let's simplify our abstraction to the 2nd level.

[2] With this type of inheritance, you need to be careful, it can lead to a diamond problem and confusion with the constructors. Remember that the tasks that are solved by multiple inheritance can be solved by other mechanisms, such as an interface.

An example of a polymorphism and inheritance problem

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;

  public:
    virtual void Attack() {
       cout << " Character attacks!" << endl;
    }
}

class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
   public:
    virtual void Attack() {
       cout << " The shaman summons a battle totem!" << endl;
    }
}

class Warrior : public Character {
    int weaponDurability;
    int weaponAttackBonus;
    int shieldPoints;
   public:
    virtual void Attack() {
       cout << " Warrior strikes with a weapon!" << endl;
    }
}
```

This is a simplified version of the Attack() function overload. Let's implement a variant when an attack affects the stats of the attacking and attacked characters using the example of a shaman.

```
class Character {
    int health;
    int attack;
    int armor;
    static size_t count;

```

```
  public:
    virtual void Attack(Character opponent) {
       cout << "Персонаж атакует!" << endl;
    }

    virtual void WereAttacked(int damage) {
       cout << "Персонаж был атакован!" << endl;
    }
}

class Shaman : public Character {
    int mana;
    int totemsHealth;
    int totemsAttack;
   public:
    virtual void Attack(Character opponent) {
       cout << "Шаман призывает боевой тотем!" << endl;
       totemsHealth = 10;
       totemsAttack = 10;
       mana--;
       opponent.WereAttacked(totemsAttack);
    }
    virtual void WereAttacked(int damage) {
       cout << "Персонаж был атакован!" << endl;
       if (totemsHealth >= demage) {
           totemsHealth -= demage;
       } else {
           if (armor == 0) {
              health -= (demage - totemsHealth);
              totemsHealth = 0;
           } else {
              if (armor >= demage - totemsHealth) {
                  armor -= (demage - totemsHealth);
                  totemsHealth = 0;
              } else {
                  health -= (demage - armor - totemsHealth);
                  armor = 0;
                  totemsHealth = 0;
              }


         }
    }
}
```

Explanations for this program: WereAttacked distributes damage - first it falls on the health of the totem, if the totem dies, it is checked whether the character has armor, if there is - the damage falls on it, if there is no more armor and the totem, but the damage remains - the damage is applied and character's health.

TASK FOR INDEPENDENT SOLUTION:

1. Create methods, RestoreHealth() like this:
   a. The shaman can only restore health at the expense of his mana. 100 mana points are converted into 10 health points.
   b. The warrior can restore health at the expense of the shield. 10 defense points are converted into 10 health points.

USE INHERITANCE!!!

Basic principles of OOP: encapsulation

Encapsulation is the control of access to the fields and methods of an object. Access control means not only possible/impossible, but also various validations, loadings, calculations and other dynamic behavior.

In many languages, part of encapsulation is data hiding. To do this, there are access modifiers (we will describe those that are in almost all OOP languages):

— public - anyone can access the attribute,
— private - only methods of this class can access the attribute,
— protected - the same as private, only the heirs of the class also get access.

How to choose the right access modifier? In the simplest case, this is how: if the method should be available to external code, we select public. Otherwise, private. If there is inheritance, then protected may be required in the case where the method should not be called from outside, but should be called by descendants.

Accessors (getters and setters)

We talked about these concepts earlier, let's take a closer look.

Getters and setters are methods whose task is to control access to fields. The getter reads and returns the value of the field, and the setter, on the contrary, takes the value as an argument and writes it to the field. This makes it possible to provide such methods with additional processing. For example, a setter, when writing a value to a field of an object, can check the type, or whether the value is within the allowed range (validation). You can add lazy initialization or caching to the getter if the actual value actually lies in the database. Many applications can be imagined.

Some languages have syntactic sugar that allows such accessors to be masked as properties, which makes access transparent to external code, which does not even suspect that it is working not with a field, but with a method that, under the hood, is running an SQL query or reading from a file. This is how abstraction and transparency are achieved.

Abstract classes

In addition to ordinary classes, some languages have abstract classes. They differ from ordinary classes in that you cannot create an object of such a class. Why is such a class needed, the reader asks? It is needed so that descendants can inherit from it - ordinary classes, whose objects can already be created.

An abstract class, along with ordinary methods, contains abstract methods without implementation (with a signature, but without code), which the programmer who wants to create a descendant class must implement. Abstract classes are not required, but they help to establish a contract that requires a certain set of methods to be implemented in order to save a programmer with a bad memory from an implementation error.

Operator overloading

Operator overloading in programming is one of the ways to implement polymorphism, which consists in the possibility of the simultaneous existence in the same scope of several different variants of using operators that have the same name, but differ in the types of parameters to which they are applied. In the publications of the Soviet period, similar mechanisms were called redefinition or redefinition, overlapping operations.

If the operator is "overloaded", then it can be used in other methods in its usual form. For example, commands for elementwise summation of two arrays a1 and a2

```
a1.add(a2);
```

```
a3 = add(a1, a2);
```

it's better to call in a more natural way:

```
a1 = a1 + a2;

a3 = a1 + a2;
```

In this example, the '+' operator is considered **overloaded**.

Operator overloading basics

First, almost any existing C++ operator can be overloaded. Exceptions are:

- ternary operator (?:);
- sizeof operator;
- scope resolution operator (::);
- member selection operator (.);
- pointer, as a member selection operator (.*).

Second, you can only overload existing operators. You cannot create new ones or rename existing ones.

Third, at least one of the operands of an overloaded operator must be of a user-defined data type. This means that you cannot overload operator+() to add an int value to a double value. However, you can overload operator+() to add an int value to an object of class MyComplexNumber.

Fourth, the initial number of operands supported by an operator cannot be changed. Those. with a binary operator, only two operands are used, with a unary operator, only one, and with a ternary, only three.

Finally, all operators retain their default precedence and associativity (regardless of what they are used for) and this cannot be changed.

Comment. Some novice programmers try to overload the bitwise XOR operator (^) to perform exponentiation. However, in C++, the ^ operator has a lower precedence than the basic arithmetic operators (+, -, *, /), and this will cause expressions to be processed incorrectly.

Rule: When overloading operators, try to keep the functionality of the operators as close as possible in accordance with their original applications.

Implementation Features

For a given class, the operator function in the class can be implemented:

— inside the class

In this case, the operator function is a class method.

— outside the classroom

In this case, the operator function is declared outside the class as "friendly" (with the **friend** keyword).

The general form of an operator function implemented in a class is as follows:

```
type operator#(arguments_list) {

    // some operations

    // ...

};
```

type - is the type of the value returned by the operator function;

operator# - is a keyword that defines an operator function in a class. The # symbol is replaced by a C++ language operator that is overloaded. For example, if the operator + is overloaded, then you need to specify operator+;

argument_list - a list of parameters that the operator function receives. If a binary operator is overloaded, argument_list contains one argument. If a unary operator is overloaded, then the argument list is empty. If the function is external and is friendly, then there can be 1 and 2 arguments, respectively. The parameters of the "friendly" function are the right operands. To refer to the left operand, a pointer to the object is used, for which the operator is called - this. Often the this pointer is omitted.

An operator function can return objects of any type. Most often, an operator function returns an object of the type of the class in which it is implemented or with which it operates.

Basic rules on the topic "Arguments and return values"

— **If the argument is not modified by the operator**, in the case of a unary plus, for example, it must be passed as a **reference to a constant**. In general, this is true for almost all arithmetic operators (addition, subtraction, multiplication...)

— The type of the returned value depends on the nature of the operator. If the operator must return a new value, then a new object must be created (as in the case of binary plus). If you want to prevent changing an object as an l-value, then you need to return it as a const.

— **Assignment statements must return a reference to the changed element.** Also, if you want to use the assignment operator in constructions like (x=y).f(), where the function f() is called for for the variable x, after assigning y to it, then don't return a reference to a constant, just return a reference.

— Logical operators should return int at worst and bool at best.

The assignment operator is necessarily defined as a class function (within the class) because it is inextricably linked to the object to the left of the "=".

An example of an assignment operator (note the approach to changing an object):

```
Integer& operator=(const Integer& right) {
    // проверка на самоприсваивание
    if (this == &right) {
        return *this;
    }
    value = right.value;
    return *this;
}
```

Comment. Checking for self-assignment is needed so as not to waste time on operations like x = x; After all, objects can have many fields, including entire arrays of data, and this operation is very labor-intensive!

TASK FOR INDEPENDENT SOLUTION:

0. First of all, read more information by overloading of operators:

https://en.cppreference.com/w/cpp/language/operators

https://www.freecodecamp.org/news/how-to-overload-operators-in-cplusplus/ (important)

1. Add assignment operator overloads operator=() to ALL classes.
2. Add an overload of comparison operators (==, <, >) to compare characters by winning chances (i.e. if both characters hit each other - who wins)
3. Add overloading of I/O operators.

Conclusion

In today's environment, having the word class in your code doesn't make you an OOP programmer. If you don't use mechanisms (polymorphism, composition, inheritance, etc.) and instead use classes just to group functions and data, then it's not OOP. The same can be solved with some data structures or namespaces. Do not confuse, otherwise you will be ashamed at the interview.

OOP in general should not be used where it makes no sense or can harm. Widespread application without knowledge of the case can cause serious architectural problems in many projects. In design, there are no unambiguous recipes for all occasions, where what is appropriate to apply, and where it is inappropriate. This will gradually fit in the head with experience.

Useful Applications

Pros and cons of OOP

Pros	**Cons**
Easy to read. There is no need to look for functions in the code and find out what they are responsible for. Consumes more memory.	Objects consume more RAM than primitive data types.
Writes quickly. You can quickly create the entities that the program needs to work with.	Reduces performance. Many things are technically implemented differently, so they use more resources.
It is easier to implement a large functionality. Since it takes less time to write code, you can create an application with many features much faster.	Difficult to start. The OOP paradigm is more complex than functional programming, so it takes more time to get started.
Less reps. No need to write the same type of functions for different entities	

FINAL TASK:

Based on the already developed blanks, **implement an interesting gameplay of the game**.