ICT 1210 – O .O. P WITH C++

Instructor: Bilenne ndifor
Email: bilenne144@gmail.com
awahbilenne.ndifor@ictuniversity.org

2020

Pokemon
Name: Pikachu
Type: Electric
Health: 70
attack()
dodge()
evolve()

Fields

Methods

PART 2
OBJECT ORIENTED PROGRAMMING

1. The Context of Software Development
2. Writing a C++ Program
3. Values and Variables
4. Expressions and Arithmetic
5. Conditional Execution
6. Iteration
7. Other Conditional and Iterative Statements
8. Using Functions
9. Writing Functions
10. Managing Functions and Data
11. Sequences
12. Sorting and Searching
13. Standard C++ Classes
14. Custom Objects
15. Fine Tuning Objects
16. Building some Useful Classes
17. Inheritance and Polymorphism
18. Memory Management
19. Generic Programming
20. The Standard Template Library
21. Associative Containers
22. Handling Exceptions

INTRODUCTION

- Now that you have the basics in C++, let us enter into the core of the course: **Object Oriented Programming (OOP) !**
- *OOP is a simply a new way of programming*.
- Classic "procedural" programming languages before C++ (such as C) often focused on the question "What should the program do next?" The way you structure a program in these languages is:
  - Split it up into a set of tasks and subtasks
  - Make functions for the tasks
  - Instruct the computer to perform them in sequence
- With large amounts of data and/or large numbers of tasks, this makes for complex and unmaintainable programs.

INTRODUCTION

- **Consider the task of modeling the operation of a car**.
- Such a program would have lots of separate variables storing information on various car parts, and there'd be no way to group together all the code that relates to, say, the wheels.
- It's hard to keep all these variables and the connections between all the functions in mind.
- To manage this complexity, it's nicer to package up self-sufficient, modular pieces of code.
- People think of the world in terms of interacting **objects**: we'd talk about interactions between the steering wheel, the pedals, the wheels, etc.
- ***OOP allows programmers to pack away details into neat, self-contained boxes (objects) so that they can think of the objects more abstractly and focus on the interactions between them.***
- In this section, we shall talk of two faces of OOP: from the *users* perspective and from the *creators* perspective.
- We shall begin by using objects and then we shall proceed on how to create objects.

INTRODUCTION

WHAT ARE OBJECTS ?

- In the physical world, we can site examples like a screen, a car, a mobile phone, …
- In a computer program, we can site examples like a window, a button, a video player, the music, …

IS AN OBJECT A VARIABLE OR A FUNCTION ?

- In reality, it's neither one nor the other.
- ***An object is a mixture of several variables and functions.***
- The variables are called ***attributes*** and the functions are called ***methods.***
- To access the method of a function, we should write: `nameObject.method`
- An example of an object we have been using so far is the ***string.***
- This can be understood by recalling the fact that a computer doesn't understand alphabets. It only understands numbers. Hence standards such as the ASCII code exist to enable the conversions.

STRINGS

- We saw how to deal with characters using the operation **`char`**.
- In order to deal with several characters, we may declare an array of characters but unfortunately, the size of the array should be fixed (static arrays).
- A solution to solve this problem was to create an object that has several member functions (methods) and variables (attributes): **`string`**.

STRINGS

```cpp
#include <iostream>
#include <string> // This is compulsory when dealing with string objects

using namespace std;

int main()
{
    string myChain1("Good morning !"); //Creating an object "myChain" of type string
    string myChain2 ("Bye Bye !"); string myChain3;

    cout << myChain1 << endl;

    myChain1 = "How are you doing ?"; //You must use the "=" sign if you wish to change
                                      //the content of a string after declaration
    cout << myChain1 << endl;

    myChain3 = "I repeat : " + myChain1 + " " + myChain2;

    cout << myChain3 << endl;

    return 0;}
```

```
Good morning !
How are you doing ?
I repeat : How are you doing ? Bye Bye !
```

STRINGS

```cpp
#include <iostream>
#include <string> // This is compulsory when dealing with string objects

using namespace std;

int main()
{
    string myChain1("Good morning !"); //Creating an object "myChain" of type string
    string myChain2 ("Bye Bye !"); string myChain3;

    if (myChain1==myChain2){

     cout<< "The sentences are the same";
    }

    else{

        cout<< "The sentences are different";
    }
    return 0;}
```

```
The sentences are different
```

STRINGS

- From what we just said and saw on strings, we can understand that strings are more complicated than we thought in the background.
- Indeed, strings involve a lot of methods (functions). Examples are:
  - The method **`size()`**
  - The method **`erase()`**
  - The method **`substr()`**
  - The ethod **`c_str()`**, …

STRINGS

```cpp
#include <iostream>
#include <string> // This is compulsory when dealing with string objects

using namespace std;

int main()
{
    string myChain("Good morning !");

    cout << myChain[3] << endl;

    cout << "Length of the chain is: " << myChain.size() <<endl;

    cout << myChain.substr(5) << endl;

    cout << myChain.substr(3, 6) << endl;

    myChain.erase();

    cout << "The chain contains : " << myChain << endl;
return 0;}
```

```
d
Length of the chain is: 14
morning !
d morn
The chain contains :
```

CLASSES: *part 1*

- Before building a house, it is necessary to come out with the plan of construction.
- In our situation, the house corresponds to the **objects** and the construction plan corresponds to the **classes**.
- Hence, an object is the complete materialization of a class just as a house is the complete materialization of the plan of construction.
- Just as we can use the same plan of construction to build several house, we can equally use a class several times to create objects of the same type.
- A class is made up of variables called **attributes** or **member variables** and functions called **methods** or **member functions**.
- By convention, we shall begin the name of a class with a capital letter. Also, we shall declare the methods before the attributes.

CLASSES: *part 1*

Creating the class

```
class Gamecharacter
{
    // Methods
    void receiveHits(int nbHits)
    {
    }
    void attack(Gamecharacter &target)
    {
    }
    void drinkLifePotion(int quantityPotion)
    {
    }
    void changeWeapon(string nameNewWeapon, int HitsNewWeapon)
    {
    }
    bool isAlife()
    {
    }
    // Attributes
    int m_life;
    int m_magicLevel;
    string m_nameWeapon;
    int m_HitsWeapon;
};
```

Remarks:

- ***we create classes to define the functioning of an object. That is what we are doing now.***
- ***Objects are created to be used. That is what we did in the previous chapter.***

CLASSES: *part 1*

Using the object

```
int main()
{
    Gamecharacter songoku, freezer; //creatin of 2 objects of type Gamecharacter

    freezer.attack(songoku); //freezer attacks songoku
    songoku.drinkLifePotion(20); //songoku increases his life by 20 (we assume the maximum life is 100
    freezer.attack(songoku); //freezer reattacks songoku
    songoku.attack(freezer); //songoku responses by an attack

    freezer.changeWeapon("50% of final transformation", 50);
    freezer.attack(songoku);

    return 0;}
```

CLASSES: *part 1*

- If we try to compile the combination of the two previous codes (creating the class and using the objects; the class should be before your main()), then we shall observe a **compilation error**.
- This is because our main function did not have the **right to access** the methods.

```
error: 'void Gamecharacter::attack(Gamecharacter&)' is private
error: within this context
```

- There are 2 main access specifiers: **Public** and **Private**.
- Public: the attributes or method can be called from exterior
- Private: the attributes or method can't be called from exterior
- By default, attributes and methods are private.
- ***Generally, we may use public with our methods but it is strictly recommended to always use private with the attributes.*** Using this approach it means we can not modify a game character's life via the main. That is referred to as **encapsulation**.

CLASSES: *part 1*

- **Encapsulation:** grouping related data and functions together as objects and defining an interface to those objects.
- Encapsulation makes such that other developers are up-larged to use only your methods.
- So far, we have our class above the main in the .cpp file. Our code seems to work correctly but it's a bit nagging.
- Let us ameliorate this by using the header file (.h) and the source file (.cpp).
- I recommend you begin by creating the files Gamecharacter.h and Gamecharacter.cpp. (File>>new >>file>>…)
- The .h file will have the class declaration together with the attributes and the method prototypes.
- ***Never use namespace in the .h file***.
- In the file Gamecharacter.cpp, we insert the code of our methods

CLASSES: *part 1*

Gamecharacter.h file

main.cpp | *Gamecharacter.h | *Gamecharacter.cpp

```
2   #define GAMECHARACTER_H_INCLUDED
3
4   #include <string>
5   class Gamecharacter
6   {
7       public:
8       void receiveHits(int nbHits)
9       void attack(Gamecharacter &target)
10      void drinkLifePotion(int quantityPotion)
11      void changeWeapon(string nameNewWeapon, int HitsNewWeapon)
12      bool isAlife()
13
14     private:
15      int m_life;
16      int m_magicLevel;
17      std::string m_nameWeapon; //DO NOT USE namespace
18      int m_HitsWeapon;
19  };
20
21  #endif // GAMECHARACTER_H_INCLUDED
```

The class functions are made public for them to access the class variables since our class variabes should be private.

CLASSES: *part 1*

Gamecharacter.cpp file ***"The format"***

main.cpp | *Gamecharacter.h | ****Gamecharacter.cpp***

```
1  #include <string> // This is compulsory when dealing with string objects
2  #include Gamecharacter.h
3
4  using namespace std;
5
6  void Gamecharacter::receiveHits(int nbHits) //this is how we write now.
7  //the compiler understands that we have a method receiveHits in the class Gamecharacter
8  {
9  }
10 void Gamecharacter::attack(Gamecharacter &target){
11 }
12 void Gamecharacter::drinkLifePotion(int quantityPotion)
13 {
14 }
15 void Gamecharacter::changeWeapon(string nameNewWeapon, int HitsNewWeapon)
16 {
17 }
18 void Gamecharacter::isAlife(){}
```

"::" IS CALLED THE BINARY SCOPE RESOLUTION OPERATOR

CLASSES: *part 1*

Gamecharacter.cpp file

Inserting the methods codes

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp

```
1  #include <string> // This is compulsory when dealing with string objects
2  #include Gamecharacter.h
3
4  using namespace std;
5
6  void Gamecharacter::receiveHits(int nbHits) //this is how we write now.
7  //the compiler understands that we have a method receiveHits in the class Gamecharacter
8  {
9      m_life -= nbHits;
10     //we decrease the character's life expectancy
11     if (m_life < 0) //to avoid getting a negative life
12     {
13        m_life = 0; //we put his life to 0n(i.e. he's dead)
14     }
15 }
16 void Gamecharacter::attack(Gamecharacter &target)
17 {
18     target.receiveHits(m_HitsWeapon);
19     //we hit the target seriously with our weapon
20
21 }
```

CLASSES: *part 1*

Gamecharacter.cpp file

continuation

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp

```
36  }
37
38  void Gamecharacter::drinkLifePotion(int quantityPotion)
39  {
40      m_life += quantityPotion;
41  if (m_life > 100) //we shouldn't go beyond 100
42  {
43     m_life = 100;
44  }
45  }
46  void Gamecharacter::changeWeapon(string nameNewWeapon, int HitsNewWeapon)
47  {
48      m_nameWeapon = nameNewWeapon;
49      m_HitsWeapon = HitsNewWeapon;
50  }
51  bool Gamecharacter::isAlife()
52  {
53      if (m_life > 0) // he's alife ?
54      {
55          return true;
56      }
57      else
58      {
59          return false;
60          }
61  }
```

CLASSES: *part 1*

The main.cpp file

The main stays the same and is even shorter now

```
#include <iostream>
#include <string>
#include "Gamecharacter.h" //Do not forget

using namespace std;

int main()
{
    Gamecharacter songoku, freezer; //creatin of 2 objects of type Gamecharacter

    freezer.attack(songoku); //freezer attacks songoku
    songoku.drinkLifePotion(20); //songoku increases his life by 20 (we assume the maximum life is 100
    freezer.attack(songoku); //freezer reattacks songoku
    songoku.attack(freezer); //songoku responses by an attack

    freezer.changeWeapon("50% of final transformation", 50);
    freezer.attack(songoku);

    return 0;}
```

DO NOT EXECUTE YOUR CODE FOR THE MOMENT, WE HAVEN'T INITIALISED THE ATTRIBUTES YET.

CLASSES: *part 2*

- **Why are our attributes not initialised?** *Because we could not do it there***.**
- **How can we initialize attributes ?** *By using* ***constructors*****.**
- If there is a constructor, then there should be a **destructor**.
- We need to use a constructor when creating attributes of the type int, double, char, …
- In C++, the constructors of object attributes such as strings are automatically created by the compiler and initialized.
- We should generally create our constructors ourselves.

CLASSES: *part 2*

HOW DO WE CREATE CONSTRUCTORS ?

- We should bare in mind the following:

A constructor is a method that respects the following 2 conditions:

- ***It has the same name as the class***
- ***It doesn't return anything, not even void.***

- Hence, to recognize a constructor in a code, we shall observe that it has the same name as the class and it returns nothing.

CLASSES: *part 2*

Prototype declaration of constructor in .h file

main.cpp | *Gamecharacter.h | *Gamecharacter.cpp

```
2   #define GAMECHARACTER_H_INCLUDED
3
4   #include <string>
5   class Gamecharacter
6   {
7       public:
8           Gamecharacter(); //Constructor
9       void receiveHits(int nbHits)
10      void attack(Gamecharacter &target)
11      void drinkLifePotion(int quantityPotion)
12      void changeWeapon(string nameNewWeapon, int HitsNewWeapon)
13      bool isAlife()
14
15     private:
16      int m_life;
17      int m_magicLevel;
18      std::string m_nameWeapon; //DO NOT USE namespace
19      int m_HitsWeapon;
20  };
21
22  #endif // GAMECHARACTER_H_INCLUDED
```

CLASSES: *part 2*

Implementation of constructor in .cpp file

```
Gamecharacter::Gamecharacter() //NO VOID
{
    m_life = 100;
    m_magicLevel = 100;
    m_nameWeapon = "0% of final transform";
    m_HitsWeapon = 10;
}
```

Remarks

- ***There is no void and no return.***
- ***With the way our variables have been declared, both game characters songoku and freezer shall start the fight with the same strengths since any time we shall create an object in the main() called `Gamecharacter`, it shall have a full life (100), full magic level, …***

CLASSES: *part 2*

Another way to implement a constructor in .cpp file: The list implementation

```
Gamecharacter::Gamecharacter():
    m_life(100), m_magicLevel(100), m_nameWeapon("0% of final transform"), m_HitsWeapon(10); //NO VOID
{
    //WE INSERT NOTING HERE, ALL HAS BEEN DONE ALREADY
}
```

Remarks

- ***There is a colon after*** `Gamecharacter()`
- ***The attributes are separated by commas and ot semi colons.***
- ***Noting changes at the level of .h file.***

CLASSES: *part 2*

OVERRIDING METHODS

- We considered by default that `Gamecharacter()` has no parameters.
- We may wish to create a player who starts a game with the best weapon. Hence, we'll have to override the method.
- This will then lead to a slight modification of the code in the files .h and .cpp

Prototype declaration of constructor in .h file

```
Gamecharacter(std::string nameWeapon, int HitsWeapon);
```

CLASSES: *part 2*

Implementation of constructor in .cpp file

```
Gamecharacter::Gamecharacter(std::string nameWeapon, int HitsWeapon) :
    m_life(100), m_magicLevel(100),m_nameWeapon(nameWeapon), m_HitsWeapon(HitsWeapon)
{

}
```

Hope you now understand why we prefixed our attributes by "m". Now, we can differentiate between the attributes and the normal variables

In the main.cpp

```
int main()
{
    Gamecharacter songoku, freezer("0% of final transform", 20);
    //creating of 2 objects of type Gamecharacter
    //creating the player freezer with the best weapon
```

If we want to create freezer with the best weapon at the start for a 0% transformation.

Exercises:

Exercise :

Recall that the user is the person who creates objects meanwhile the conceiver is the person who creates classes.

Permit the user to modify the life and the magic level at the start.

Hint: you'll require a third constructor

CLASSES: *part 2*

DESTRUCTORS

- A destructor is used to delete an object in memory.
- It helps in deallocating the memory (via **`delete`**) if he memory was allocated (via **`new`**).
- The destructor must have:
  - the same name as the class, but preceded with a tilde sign (~),
  - it must also return no value,
  - and it must take no parameter.
- In our example, we do not need a destructor since we used no allocation.

```
~Gamecharacter(); //Destructor
```

In the .h file

```
Gamecharacter::~Gamecharacter()
{
//WE INSERTED NOTHING SINCE WE DO NOT NEED A DESTRUCTOR
}
```

In the .cpp file

CLASSES: *part 2*

CONSTANT METHODS

- Constant methods are methods to “read” only. i.e., one can only view what it displays and can not modify it.
- It is indicated by adding the key-word **`const`** at the end of the prototype and the declaration.
- When we tell the compiler that the “method is constant”, it means that the method doesn’t modify the object i.e. it doesn’t modify the attribute.
- E.g. a method that displays the life level of the game character can not be constant meanwhile a method that displays general information on the game character is a constant.

```
//Prototype of the method (in .h) :
void myMethod(int parameter) const;


//Declaring the method (in .cpp) :
void MaClasse::maMethode(int parametre) const
{

}
```

In the .h file

In the .cpp file

CLASSES: *part 2*

CONSTANT METHODS

- With the string example, one example of a constant method is the size method.
- In our example, we can talk of the method isAlife.

```
bool Gamecharacter::isAlife() const
{
    if (m_life > 0) // he's alife ?
    {
        return true;
    }
    else
    {
        return false;
        }
}
```

In the .cpp file

CLASSES: *part 2*

ASSOCIATING CLASSES AMONG THEMSELVES

- OOP becomes very interesting and useful when one can combine several objects with each other.
- With our example, we created the class **`Gamecharacter`**. If instead of putting information about the weapons in the class **`Gamecharacter`**, one can think of creating a separate class named Weapon.
- Hence, now I hope you have started reasoning in an object oriented manner.

NB: there are hundred of ways of creating an object oriented program. Everything lies on the organization of classes, the way they interact, etc.

- To create a new class, go to File>>New>>Class
- Creating a new class obviously leads to the creation of the files **`Weapon.h`** and **`Weapon.cpp`** (we do this for organizational reasons, we can also decide to put everything in one file).

CLASSES: *part 2*

In the `Weapon.h` *file*

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp | ***Weapon.h** | Weapon.cpp

```
1  #ifndef WEAPON_H_INCLUDED
2  #define WEAPON_H_INCLUDED
3
4  #include <iostream>
5  #include <string>
6
7  class Weapon
8  {
9      public:
10
11     Weapon();
12     Weapon(std::string name, int hits);
13     void change(std::string name, int hits);
14     void display() const;
15
16     private:
17
18     std::string m_name;
19     int m_hits;
20 };
21 #endif // WEAPON_H_INCLUDED
```

CLASSES: *part 2*

In the `Weapon.cpp` *file*

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp | *Weapon.h | **Weapon.cpp**

```
1   #include "Weapon.h"
2
3   using namespace std;
4
5   Weapon::Weapon() : m_name("0% of final transform"), m_hits(10)
6   {
7   }
8
9   Weapon::Weapon(string name, int hits) : m_name(name), m_hits(hits)
10  {
11  }
12
13  void Weapon::change(string name, int hits)
14  {
15      m_name = name;
16      m_hits = hits; }
17
18  void Weapon::display() const
19  {
20      cout << "Weapon : " << m_name << " (Hits : " << m_hits << ")" << endl;}
```

CLASSES: *part 2*

ASSOCIATING CLASSES AMONG THEMSELVES

- With all this done, what is left is for us to adapt the **`class Gameplayer`** for him to use the class Weapon.

In the .h file:

- we start by removing the attributes **`m_nameWeapon`** and **`m_hitsWeapon`** (since they're now useless) and we add **`#include "Weapon.h"`**
- **DO NOT TOUCH THE METHODS** (since they're already used by a user in the main(), hence any modification can make our program not to function).

Generally, we should avoid using public at the level of variables for reasons we said earlier and we should avoid changing the name of our methods for reasons said above.

CLASSES: *part 2*

In the `Gamecharacter.h` *file*

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp | *Weapon.h | Weapon.cpp

```
1  #ifndef GAMECHARACTER_H_INCLUDED
2  #define GAMECHARACTER_H_INCLUDED
3
4  #include <string>
5  #include "Weapon.h"
6
7  class Gamecharacter
8  {
9      public:
10         Gamecharacter(); //Constructor
11         Gamecharacter(std::string nameWeapon, int HitsWeapon);
12
13        ~Gamecharacter(); //Destructor
14
15     void receiveHits(int nbHits)
16     void attack(Gamecharacter &target)
17     void drinkLifePotion(int quantityPotion)
18     void changeWeapon(string nameNewWeapon, int HitsNewWeapon)
19     bool isAlife()
20
21     private:
22         int m_life;
23         int m_magicLevel;
24         Weapon m_weapon; //Our game player has a weapon
25 };
26
27 #endif // GAMECHARACTER_H_INCLUDED
```

CLASSES: *part 2*

In the .cpp file:

- We just modify all what was influenced by our former variables (those highlighted in yellow)

Previous code in the `Gamecharacter.cpp` ***file***

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp | *Weapon.h | Weapon.cpp

```
4   using namespace std;
5
6   Gamecharacter::Gamecharacter():
7       m_life(100), m_magicLevel(100), m_nameWeapon("0% of final transform"), m_HitsWeapon(10)   //NO VOID
8   {
9       //WE INSERT NOTING HERE, ALL HAS BEEN DONE ALREADY
10  }
11
12  Gamecharacter::Gamecharacter(std::string nameWeapon, int HitsWeapon) :
13      m_life(100), m_magicLevel(100),m_nameWeapon(nameWeapon), m_HitsWeapon(HitsWeapon)
14  {
15
16  }
```

CLASSES: *part 2*

Previous code in the `Gamecharacter.cpp` file

*main.cpp | *Gamecharacter.h | *Gamecharacter.cpp | *Weapon.h | Weapon.cpp

```
4   using namespace std;
5
6   Gamecharacter::Gamecharacter():
7       m_life(100), m_magicLevel(100), m_nameWeapon("0% of final transform"), m_HitsWeapon(10)   //NO VOID
8   {
9       //WE INSERT NOTING HERE, ALL HAS BEEN DONE ALREADY
10  }
11
12  Gamecharacter::Gamecharacter(std::string nameWeapon, int HitsWeapon) :
13      m_life(100), m_magicLevel(100),m_nameWeapon(nameWeapon), m_HitsWeapon(HitsWeapon)
14  {
15
16  }
```

Present code in the `Gamecharacter.cpp` file

```
Gamecharacter::Gamecharacter():
    m_life(100), m_magicLevel(100)
{
    //WE INSERT NOTING HERE, ALL HAS BEEN DONE ALREADY
}

Gamecharacter::Gamecharacter(std::string nameWeapon, int HitsWeapon) :
    m_life(100), m_magicLevel(100),m_Weapon(nameWeapon, HitsWeapon)
{

}
```

CLASSES: *part 2*

How can we access the variables

- Generally, since our attributes are private, we can not access them from the main. On the other hand, any programmer who respects his/her self shouldn't declare them as public.
- **To solve this problem, we create a public method that will have as duty to access the attribute.**
- This method is referred to as the **accessor** and for identification purposes, we write the name of the method while beginning with the prefix "**get**" and it is recommended that we precise **const**.
- If we wished to instead change the value of an attribute, we generally create a method with prefix "**set**".
- All these things should be done for security reasons.

CLASSES: *part 2*

We add this code in Weapon.cpp *file*

```
int Weapon::getHits() const
{
    return m_hits;
}
```

Declaration of variable in Weapon.h *file*

```
class Weapon
{
    public:

    Weapon();
    Weapon(std::string name, int hits);
    void change(std::string name, int hits);
    void display() const;
    int getHits() const;

    private:

    std::string m_name;
    int m_hits;
};
```

CLASSES: *part 2*

Hence now we can do the following modifications

We do this code in Gamecharacter.cpp *file*

```
void Gamecharacter::attack(Gamecharacter &target)
{
    target.receiveHits(m_Weapon.getHits);
    //we hit the target seriously with our weapon
}
```

Hence from what we can observe, we called the method change when in Gamecharacter.cpp via the object `m_Weapon` just because we declared `m_Weapon` in `Gamecharacter.h` as an object from the class `Weapon`. We could do this sin ce we included `Weapon.h`

```
void Gamecharacter::changeWeapon(string nameNewWeapon, int HitsNewWeapon)
{
    m_Weapon.change(nameNewWeapon, HitsNewWeapon);
}
```

CLASSES: *part 2*

- **As we can see, it is possible to make objects interact among themselves but we have to organize them well and at every instant, we should ask ourselves the question: do I have the right to access this element or not ?**
- **Please, do not hesitate to create an accessor if need be though it may make your work bulkier.**
- **Do not make your attributes public to simplify the problem. By so doing, you loose all the security advantages offered by OOP.**

CLASSES: *part 2*

- Our game characters are fighting in the main() but we can't view this.
- To solve this problem, let us create a method **`displayState`** in **`Gamecharacter.h`** that displays to the screen the life, magic level, … of each game character.

We do this in Gamecharacter.h *file*

```
void receiveHits(int nbHits)
void attack(Gamecharacter &target)
void drinkLifePotion(int quantityPotion)
void changeWeapon(string nameNewWeapon, int HitsNewWeapon)
bool isAlife()
void displayState() const
```

Now, we can compile our code after some slide adjustments

```
Songoku
Life : 40
Magic Level : 100
Weapon : 0% of final transform (Hits : 10)

Freezer
Life : 90
Magic Level : 100
Weapon : 100% of final transform (Hits : 40)
```