

Functional (FP) vs Object Oriented (OOP)

University System Example (FP)

- AddStudent
- UpdateStudent
- DeleteStudent
- CalculateAverage
- AddCourse
- UpdateCourse
- DeleteCouse
- EnrollStudentInCourse
- UnEnrollStudentFromCoruse
- HowManyStudentsInCourse
- Doctor (Add, Edit, Delete)
- AssignCourseToDoctor
- SendEmailToStudent
- SendTextMessageToStudent
- SendEmailToDoctor
- SendTextMessageToDoctor
- CallStudent
- CallDoctor
- AddEmployee
- UpdateEmployee
- DeleteEmployee
- CalculateSalary
- PaySalary
-



Simply

You can have thousands of functions in your System!!!!



05:18

Class came from Classification



Class Members



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

class clsPerson
{
public:

string FirstName;
string LastName;

string FullName()
{
    return FirstName + " " + LastName;
}
};

int main()
{
    clsPerson Person1;

    Person1.FirstName = "Mohammed";
    Person1.LastName = "Abu-Hadhoud";

    cout << "Person1: " << Person1.FullName()
    << endl;
}
```

Members



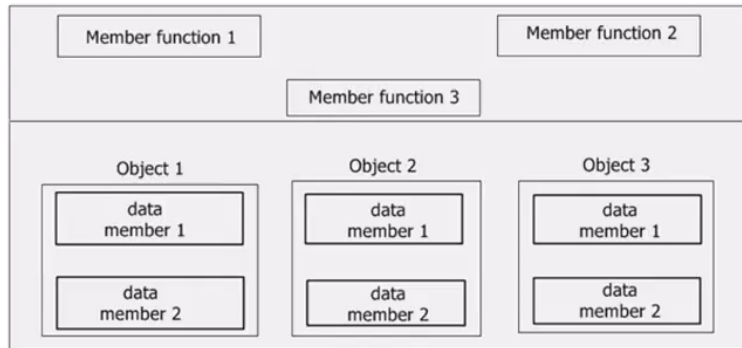
Data Members: Any variable declared inside the class that holds Data.

In Our Case:
FirstName, LastName
Are Data Members

Member Methods(Functions): Any Function Or Procedure declared inside the class.

In Our Case:
FullName()
Is a Member Function/Method

Each instance has its own space in memory, Only Member functions are shared among all objects



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

class clsPerson
{
public:
    string FirstName;
    string LastName;

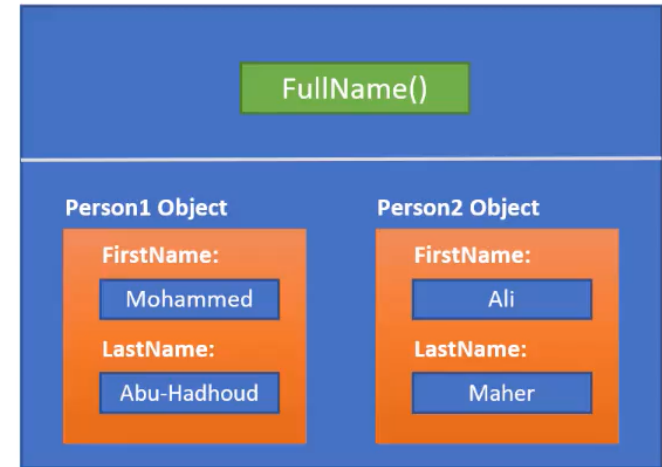
    string FullName()
    {
        return FirstName + " " + LastName;
    }
};

int main()
{
    clsPerson Person1, Person2;

    Person1.FirstName = "Mohammed";
    Person1.LastName = "Abu-Hadhoud";

    Person2.FirstName = "Ali";
    Person2.LastName = "Maher";

    cout << "Person1: " << Person1.FullName() << endl;
    cout << "Person2: " << Person2.FullName() << endl;
}
```



The `sizeof` operator in C++ calculates the size of objects and types based on their memory representation. However, it does not include the size of member functions or member function pointers in its calculation.

Member functions in a class are not part of the object's memory representation. They are typically shared among all instances of the class and exist only once in memory, regardless of the number of objects created. Each object only stores the data members (variables) of the class, not the member functions.

The size of a class object, as calculated by `sizeof`, includes only the memory required to store the data members of the class and any additional padding or alignment requirements. It does not include the memory required for member functions.

Therefore, when you use `sizeof` with a class or an object, it provides the size of the data members and any additional memory requirements imposed by the implementation and platform, but it does not include the size of the member functions themselves.

Size of Classes and Objects in Memory

- The size of a class in memory is determined by the memory layout and alignment requirements of its member variables. It can be affected by factors such as padding and memory alignment.
- The size of an object of a class includes the memory required to store its data members, any additional padding or alignment requirements, and potentially the memory needed for virtual function tables (vtables) if the class has virtual functions.
- The sizeof operator in C++ can be used to calculate the size of an object or class. It returns the size in bytes and provides an estimation of the memory required by the object or class.
- The sizeof operator calculates the size of the data members within the class. It does not include the size of member functions or virtual function tables (vtables) in its calculation.
- The actual size of an object or class may be larger than the sum of its data members due to padding and alignment requirements imposed by the compiler and platform. Padding is added to ensure proper alignment of the data members in memory.
- The size of an object or class can be influenced by various factors, including the size of its data members, alignment requirements, padding, compiler optimization settings, and platform-specific considerations.
- The sizeof operator provides a way to estimate the memory footprint of an object or class, but the actual memory usage may vary depending on the specific implementation, compiler, and platform.
- It is important to note that the sizeof operator does not account for dynamic memory allocation or additional memory overhead caused by virtual inheritance or virtual base classes.
- To accurately measure the memory usage of a program or analyze the memory layout of a class, specialized profiling tools and memory analyzers can be used.

Access Specifiers (Modifiers)



10:07



Copyright© 2022
ProgrammingAdvices.com

Muhammad Ali-Hamid
MBA, PMOC, PMP, NP®, CM, ITIL, MCSE
26+ years of experience

12:48



```

#include <iostream>
using namespace std;
class clsPerson
{
private:
    // only accessible inside this class
    int Variable1 = 5;

    int Function1()
    {
        return 40;
    }

protected:
    // only accessible inside this class and all classes inherits this class
    int Variable2 = 100;
    int Function2()
    {
        return 50;
    }

public:
    // accessible for everyone outside/inside/and classes inherits this class
    string FirstName;
    string LastName;

    string FullName()
    {
        return FirstName + " " + LastName;
    }
    float Function3()
    {
        return Function1() * Variable1 * Variable2;
    }
};

int main()
{
    clsPerson Person1;
    Person1.FirstName = "Mahmoud";
    Person1.LastName = "Mattar";

    cout << "Person1: " << Person1.FullName() << endl;
    cout << Person1.Function3();
}
/*

```

Access Specifiers:

In C++, access specifiers control the visibility and accessibility of class members. They determine which parts of a class are accessible from different contexts.

- private: The private access specifier restricts access to class members to be only within the class itself. They are not accessible from outside the class or from derived classes. Private members are typically used for internal implementation details that are not intended to be accessed directly.

- protected: The protected access specifier allows access to class members within the class itself and any derived classes.

They are not accessible from outside the class hierarchy. Protected members are often used to provide access to derived classes

while still restricting access from other parts of the program.

- public: The public access specifier provides unrestricted access to class members from any part of the program.

Public members can be accessed directly from outside the class and are inherited by derived classes.

They are typically used for interface and functionality that needs to be accessible to other parts of the program.

By default, if no access specifier is specified, members are considered private.

It's important to carefully choose the appropriate access specifier for class members based on the desired level of encapsulation


and the intended usage of the class.

Note: The access specifiers only affect the visibility of class members;

they do not impact the memory layout or size of the class or object in memory.

*/

```
1 #include <iostream>
2 using namespace std;
3
4 class clsPerson
5 {
6 private:
7     string _FirstName;
8     string _LastName;
9
10 public:
11     // Property Set: Sets the value of the first name
12     void setFirstName(string FirstName)
13     {
14         _FirstName = FirstName;
15     }
16
17     // Property Get: Retrieves the value of the first name
18     string FirstName()
19     {
20         return _FirstName;
21     }
22
23     // Property Set: Sets the value of the last name
24     void setLastName(string LastName)
25     {
26         _LastName = LastName;
27     }
28
29     // Property Get: Retrieves the value of the last name
30     string LastName()
31     {
32         return _LastName;
33     }
34
35     // Returns the full name by concatenating the first and last names
36     string FullName()
37     {
38         return _FirstName + " " + _LastName;
39     }
40 };
41
42 int main()
43 {
44     clsPerson Person1;
45     Person1.setFirstName("Mahmoud");
46     Person1.setLastName("Mattar");
47
48     cout << "First Name: " << Person1.FirstName() << endl;
49     cout << "Last Name: " << Person1.LastName() << endl;
50     cout << "Full Name: " << Person1.FullName() << endl;
51
52     system("pause > nul");
53     return 0;
54 }
```



1 In object-oriented programming, Property Set and Get methods are used to encapsulate the access and manipulation of private member variables (properties) of a class.

2 They provide a controlled way to set and retrieve the values of these variables.

3

4 explanation of Property Set and Get:

5

6 Property Set (Setter):

7

8 A Property Set method is responsible for setting the value of a private member variable.

9 It usually takes a parameter that represents the new value to be assigned.

10 Inside the setter method, you can perform any necessary validation or additional logic before assigning the value to the member variable.

11

12 Property Get (Getter):

13

14 A Property Get method is used to retrieve the value of a private member variable.

15 It does not take any parameters.

16 The getter method returns the current value of the member variable.

17

18 By using Property Set and Get methods, you can enforce encapsulation and control the access to the internal state of an object.

19 This approach allows you to protect the integrity of the object's data and provide a standardized way for other code to interact with the object's properties.

20

21 In the provided code, the Property Set methods (setFirstName and setLastName)

22 are used to set the values of the private member variables _FirstName and _LastName, respectively.

23 The Property Get methods (FirstName and LastName) are used to retrieve the values of these member variables.

24

25 Using Property Set and Get methods helps promote encapsulation and maintain the principle of information hiding within a class.