OBJECT ORIENTED PROGRAMMING

LAB 3

- POLYMORPHISM
- OPERATOR OVERLOADING
- CONSTRUCTOR WITH PARAMETERS
- OBJECT DECLARATION VS CONSTRUCTOR USED
- RESOLUTION OPERATOR
- IMPLICIT DESTRUCTOR

In the prior lab we saw that we can initialize with values the attributes of a class in two ways:

- by using an implicit constructor or
- by using special class methods called setters.
- Setting the attribute directly is not recommended in C++ as it is customary to set the attributes with the access modifier private

In this lab we will begin by explaining the constructor with parameters, which is another way to set values to class attributes. No matter what way we choose to give values to the attributes, we return the values of those attributes through getter methods.

A few of the concepts used in this session are presented below.

Polymorphism

This is a C++/OOP concept on which the compiler is picking the function variant to be used in a particular function call based on the name, existence, number and order of types for function arguments (also known as function parameters) is called POLYMORPHISM or FUNCTION OVERLOADING.

Through this concept a **function, in general**, is recognized as distinct not just through its name, but via a combination of *name, argument types* and their *order*.

Note: we did not say *return type* in this enumeration!

Function declaration example	Explanation	C++ allowed
`void SetName(string N);` `void GetName(string N);`	The 2 functions differ by **name**	✓

`void SetName();`
`//⇔void SetName(void)`
`void SetName(string N);`
	The 2 functions differ **by number and type** of function arguments	✓

`Student (int V);` `Student (int V, string N);`	The 2 functions differ **by number and type** of function arguments	✓
`Student (int V);` `Student (string N);`	The 2 functions differ by **type** of function arguments	✓
`Student (int V, string N);` `Student (string N, int V);`	The 2 functions differ by **order** of function arguments	✓
`int GetName(string N);` `string GetName(string N);`	The 2 functions differ only by return type. This is not allowed as parameter list and name are the same. C++ error is *ambiguation of old declaration*.	✗

The concept of polymorphism is used in this laboratory because we can define for a class both constructors with 0 parameters (also called implicit constructors) as well as constructors with more than 0 parameters (also called constructors with parameters).

Operator overloading

The concept of polymorphism is related to OPERATOR OVERLOADING, a concepts that allows a given operator to have different functionality based on the context it is used in. We will simply illustrate its usage in this session.

For example the + operator has a different modus operandi in the following situations.

`int a=5, b=3;` `cout << a+b <<endl;`	+ operator adds two numbers to obtain a result

`string d="My", e="name";`
`string f="is", g="John";`
`cout <<d+" "+e+" "+f+" "+g<<endl;`
	+ operator does string concatenation

Try executing the following program in VPL.

```
#include <iostream>

using namespace std;

int main(){
    int a=5,b=3;
    cout << a+b <<endl;

    string d="My", e="name", f="is", g="John";
    cout <<d+" "+e+" "+f+" "+g<<endl;

    return 0;
}
```

Constructor with parameters

For a class with 1 attribute we can have constructors:

- with 0 parameters (or arguments, also called implicit constructor), in which the single attribute will be initialized to the right value. If no constructor (implicit or with parameters) is provided in the source code, an implicit one will be generated by the compiler, zero-ing out the attributes.
- with 1 parameter, in which the same attribute will be initialized with the value passed through the function argument to the constructor with parameter.

Class attributes	Possible constructors	Object calling the constructor
`class Balloon {` `  string Color;`	`//implicit constructor` `Balloon () { Color = "Red"; }`	`//calling implicit constructor` `Balloon B1;`
	`//constructor with 1 parameter` `Balloon (string C) {Color = C;}`	`//calling constr. with 1 param.` `Balloon B2("Blue");`

Class attributes

`class Student {`
`  int Grade;`
`  string Name;`

For a class with 2 attributes we can have:

- constructor with 0 parameters (implicit constructor)

Definition of object from function main ()	(Implicit) constructor called
`Student s1;`	`Student(); //implicit constructor`

- constructors with 1 parameter (of type of each of the 2 attributes)

Definition of object from function main ()	Prototype of constructor with parameters called

`Student s1(5); /* 5 is integer so we would need a numeric data type for the parameter in the constructor */`	`Student(int); //constructor with an //integer argument (int)`
	
	`/* other viable candidates for type in constructor parameter could have been:`
	`* integer type: char, short, long, long long and also`
	`* fractional types: float, double, long double because fractional types include integer types */`

`Student s2("Joe"); /* Joe is a word, so we will need a string or char * parameter in a corresponding constructor */`	`Student (string); //constructor with a // word argument (string)`
	
	`/* other viable candidates could have had arguments of type: char * */`

- constructors with 2 parameters (of type of the 2 arguments, in all possible orders)

Definition of object from function main ()	Prototype of constructor with parameters called

`Student s3 (5, "Joe");`
`// 5 is first, //integer`
`// Joe is second, word`
	`Student (int, string);//constructor with //2 arguments`
	`// first int`
	`// then string`

`Student s3 ("Joe", 5);`
`// Joe is first, word,`
`// 5 is second, integer`
	`Student(string, int);//constructor with //2 arguments`
	`// first string`
	`// then int`

Similarly for classes with N, more than 2, attributes.

In all these situations the **constructor must initialize all attributes**, both those for which it received through the method argument an implicit value, as well as for the rest, for which it hasn't. In this case, the programmer is responsible for the initialization of remaining attributes from the method source code.

Definition of object	Prototype of constructor with parameters called	Constructor definition

`Student s1(5);`
`//5 is integer`
	`Student(int);`	`Student (int G) {`
		`/* all attributes must be initialized */`
		
		`/* parameter is fed into corresponding attribute */`
		`Grade= G;`
		
		`/* if we got no indication for this parameter upon object creation we add an implicit value */`
		`Name = "Bob";`
		`}`

`Student s2("Joe");`
`//Joe is a word`
	`Student (string);`	`Student (string N) {`
		`/* all attributes must be initialized */`
		
		`/* if we got no indication for this parameter upon object creation we add an implicit value */`
		`Grade= 1;`
		
		`/* parameter is fed into corresponding attribute */`
		`Name = N;`
		`}`

`Student s3 (5, "Joe");`	`Student(int, string);`	`Student (int G, string N) {` `/* all attributes must be initialized */`

// 5 is first,
//integer
// Joe is second,
word
	/* constructor with 2 arguments
	* first int
	* then string */
		/* parameters are fed into corresponding attribute */
		Grade= G;
		Name = N;
		}

Student s3 ("Joe", 5);
// Joe is first,
word,
// 5 is second,
integer
	Student(string, int);
	/* constructor with 2 arguments
	* first string
	* then int */
		Student (string N, int G) {
		/* all attributes must be initialized */
		
		/* parameters are fed into corresponding attribute */
		Grade= G;
		Name = N;
		}

So far the (known) means by which to set the value of an attribute consist of: the implicit constructor, constructors with parameters and setter.

Resolution operator

For ease of reading the source code, it is customary for the methods (member functions of classes) to be declared in class as prototypes, but to be defined outside the class. In this case the method names are preceded by the class name they belong to followed by the resolution operator (: :) to describe belonging to a certain class and distinguish from regular non-class member functions.

To specify outside of a class the belonging of a method we first specify function return type, then class name, resolution operator (::), name of method, followed by the list of types and names of the method.

```
<Function_return_type > <Class_name > :: <Method_name> (<Argument_type1>
<Argument_name1>,  <Argument_type2> <Argument_name2>, …)
```

Between the method declaration in the class and the definition outside the class there is a correspondence between the function elements. Let's use as an example the class *SetMaxNrNails* in class *NailGun*.

Declaration	Definition
Specifying the name (ex: N) of the method (ex:SetName) argument is **optional** in declaration. This makes it correct not to specify in the prototype of the name of the argument (N).	Specifying the name (ex: N) of the method (ex:SetName) argument is **mandatory** in definition, as the argument will be used in the method body.

class **NailGun** {
…
//prototype aka declaration
void SetMaxNrNails(**int**);
…
};
	//definition
	void NailGun::**SetMaxNrNails**(**int** N) {*MaxNrNais* = N; }

• Class name NailGun • Returns void • Method called SetMaxNrNails • First argument (parameter) of type int • Optional to specify name of argument (parameter) in declaration (prototype)	• Returns void • To class NailGun • It belongs (::) • And is called SetMaxNrNails • Has a first argument of type int • **Argument is called N. Mandatory to provide argument name in definition.** • ***MaxNrNails* is the name of the attribute from class Student** • **Function also has a body**

Implicit destructor

For any object we also have a destructor, responsible to free the resources allocated for the object before we exit the scope in which it was used.

The destructor is a special method in the class. It's name is the class name preceded by ~. In case of more complicated classes, in which the constructor also allocated memory, it's the destructor's job to free those resources.

Throughout our labs the destructor will mostly be empty (there are no malloc's to be freed).

Class name	Implicit destructor prototype	Implicit destructor definition (in our sessions)
class Student { …	~Student();	~Student () {}

Bool

Is a type of data in C++. It can hold the values true (1) or false (0). Generically in conditional statements any non-zero result is considered true and 0 is considered false.

EXAMPLE 1

We model a new data type via the 'NailGun" class. This new data type will contain as attributes:

- ✓ The maximum number of nails held by the nail gun, stored in an int type variable
- ✓ The current number of nails, between 0 and max, also stored in an int type variable
- ✓ A safety with which we can block the nail gun from firing, stored in a Boolean type variable (the safety can be either ON or OFF)

We are going to create the following methods:

- ✓ A setter and a getter for the maximum number of nails and the safety
- ✓ A getter for the current number of nails (we can't set the current number of nails held by the nail gun, so we won't implement a setter for that attribute)
- ✓ An implicit constructor - used to initialize the maximum number of nails and the state of the safety.
    - o When creating an object without arguments (parameters)
    - o With a name identical with the class name (case sensitive)
    - o If we would not specify in our program any constructor one would be created by the compiler on build
- ✓ A constructor with parameters
    - o For the purpose of initializing the object attributes to values specified outside the class. For our examples the constructors will not allocate additional resources
    - o With a name identical with the class name (case sensitive)
    - o Without a return type prior to the method name
    - o Upon object creation, the compiler analyzes the arguments; if the object to be created has no arguments then it uses the implicit constructor. In this case the object is initialized with the values from the implicit constructor
    - o If the object to be created has a set of arguments specified, then the compiler picks the constructor with the same number (and type and in the same order) of parameters as arguments. If there is no such constructor we will get an error.

- The concept used by the compiler in picking the type of constructor based on the number of arguments specified is called **POLYMORPHISM or FUNCTION OVERLOADING** (which is a different concept than OPERATOR OVERLOADING – in which an operator has a different behavior if it is used on different types of data)

✓ An implicit destructor:
- For the purpose of freeing resources allocated by the constructor
- Without a return type prior to the method name (and none to be provided by the compiler)
- If we would not specify a destructor, one would be created by the compiler on build
- With a name the name of the class (case sensitive) prefixed by the character tilda ~

✓ The Fire() method in which we first check the state of the safety; only if it is OFF we check if we still have nails and if we do we can fire the nail gun (which from the C++ class perspective means decreasing the current number of nails by 1). We will also print the message "FIRE!" when we fire the nail gun.

✓ The Load() method is called when there aren't any nails left in the nail gun. For this method, we first check if the nail gun is not already loaded. If we have less than the maximum number of nails, then we can load the nail gun. The Load() will then increase the current number of nails to its max value.

✓ The SetSafety() method first checks if the safety is not already set, and then sets the safety ON.

✓ The ReleaseSafety() method will check if the safety was already released, and only after that set the safety to OFF.

For the purpose of the example, some of the methods will be defined outside the class scope by using the resolution operator (::). The class scope is delimited by the "{" and "};" .

Let's analyze the implementation for example 1:

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

class NailGun //define the new data type
{
    int MaxNrNails;
    int CrtNrNails;
    bool Safety;

public:
    void SetMaxNrNails(int Max); //setter for MaxNrNails
    /* CrtNrNails doesn't have to have a setter because it will get incremented or decremented
by the way we call Load() to load the NailGun, or Fire() to fire the NailGun */


    void SetSafety(bool p); //setter for Safety

    int GetMaxNrNails (); //getter for MaxNrNails
    int GetCrtNrNails (); //getter for CrtNrNails
    bool GetSafety(); //getter for Safety

    void Fire();
    void Load();
    void SetSafety();
    void ReleaseSafety();

    NailGun();//implicit constructor
    NailGun(int NrMAX, bool sft);//constructor with arguments
    ~NailGun() {} //implicit destructor

};
```

```
void NailGun:: SetMaxNrNails(int Max) //setter for SetMaxNrNails
{
    MaxNrNails =Max;
}

void NailGun::SetSafety(bool p) //setter for Safety
{
    Safety=p;
}

int NailGun::GetMaxNrNails () //getter for GetMaxNrNails
{
    return MaxNrNails;
}

int NailGun:: GetCrtNrNails () //getter for  CrtNrNails
{
    return CrtNrNails;
}

bool NailGun::GetSafety() //getter for Safety
{
    return Safety;
}

void NailGun::Fire()//implementation of the fire method
{
    if(Safety==true)//check of safety already set
    {
        cout<<"Safety set, can't fire.";
    }
    else//if Safety not set - check the current number of nails
    {
         if(CrtNrNails>0)//check if I have any nails
         {
             CrtNrNails--;//fire a shot - decrease the bullet count in the NailGun
             cout<<"FIRE !!!";//fire message
         }
         else
         {//if no more nails print a message
             cout<<"Out of nails!!!";
         }
    }
}

void NailGun::Load()
{//check first if the NailGun is already loaded
    if (CrtNrNails==MaxNrNails)
    {
         cout<<"NailGun is already loaded !!!";
    }
    else//if there are less nails than max
    {
         CrtNrNails=MaxNrNails;//full load
    }
}

void NailGun::SetSafety ()
{//check if safety already set
    if(Safety==true)
    {
         cout<<"Safety already set.";
    }
```

```
    else//if safety not set, then set it
    {
         Safety=true;
    }
}

void NailGun::ReleaseSafety()
{//check if safety is already released
    if(Safety==false)
    {
        cout<<"Safety already released.";
    }
    else//if it not already released, release the Safety (put it to false)
    {
         Safety=false;
    }
}

NailGun::NailGun()
{//initialize values via the implicit constructor
    cout<<"Using the implicit constructor \n";
    MaxNrNails=5;
    Safety=false;
}

NailGun::NailGun(int NrMAX, bool sft)
{//initialize values via the constructor with 2 arguments
    cout<<"Using the constructor with 2 arguments. \n";
    MaxNrNails=NrMAX;
    Safety=sft;
}

int main()
{

    NailGun p0; /* create a NailGun type object p0, without arguments, so using the implicit
constructor */


    if(p0.GetSafety()==true)//check safety
    {//print different messages based on state of Safety
        cout << "Implicit NailGun p0 has a max of "
             << p0.GetMaxNrNails() << " nails and safety set. \n\n";
    }
    else{
        cout << "Implicit NailGun p0 has a max of "
             << p0.GetMaxNrNails()<<" nails and safety released. \n\n";
    }

    NailGun p1(10,true);/* create another NailGun type object (with arguments matching my
definition in both number and type) using the defined constructor with 2 arguments */

    if(p1.GetSafety()==true)
    {//print different messages based on state of Safety
        cout << "Implicit NailGun p1 has a max of "
             << p1.GetMaxNrNails()<<" nails and safety set. \n\n";
    }
    else{
        cout << "Implicit NailGun p1 has a max of " << p1.GetMaxNrNails()
             << " nails and safety released. \n\n";
       }
}
```

Homework

3.1. Implement a modified example in which you also have a constructor with 3 parameters, to initialize all three attributes of the class NailGun.

3.2. How should the declaration of constructor with parameters look like for class Firm, with attributes Name (of type word) and Revenue (a fractional number) for objects f1 and f2?

Definition of object in function main ()	Prototype (declaration) of constructor with parameter(s) called
`Firm f1(51200.0);`	a) `Firm(int);` b) `Firm(float, string);` c) `Firm(float);` d) `Firm(string);`
`Firm f2("SC ELearning SRL", -471.56);`	a) `Firm(string,int);` b) `Firm(float, string);` c) `Firm(float);` d) `Firm(string, float);`

3.3. Write a program to implement class Firm, with:

- attributes Name(of type word) and Revenue (of type fractional number)
- setters and getters for attributes
- implicit constructor (setting Revenue to 100, Name to "SC ACME SRL") and destructor
- constructors with proper parameters to be able to run the following code in main()

```
int main()
{
   cout << "create firm1" << endl;
   Firm firm1;

   cout << "create firm2" << endl;
   Firm firm2(51200.0);

   cout << "create firm3" << endl;
   Firm firm3("SC ELearning SRL", -471.56);

   cout << "at the end " << endl;
   return 0;
}
```

Note: it is up to you if the standard methods display a message when they are passed through.

EXAMPLE 2

This class has the **private** attributes color and tank (capacity). We create setter and getter methods for both attributes. The methods are declared inside the class as a prototype but are defined outside the class.
In this case we use the scope resolution operator (::) to describe the method belonging to the Car class.

In the main() function we create more than one Car type objects. Some of these get their values from the implicit constructor, some from the constructor with parameters. The decision to pick which constructor is taken by the compiler through checking the number of arguments and their types versus the ones for the defined constructors. The decision can be made due to the presence of polymorphism concept in C++ functions.

The source code for Car class is the following.

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

class Car
{//private attributes for class Car
    string color;
    int capacity;

public:
    void SetColor(string c);//setter for color
    void SetCapacity(int r);//Setter for tank capacity

    string GetColor();//getter for color
    int GetCapacity();//getter for tank capacity

    Car();//implicit constructor
    Car(string col, int cap);//constructor with arguments
    ~Car() {} //implicit destructor
};

void Car::SetColor(string c){
    color=c;
}

void Car::SetCapacity(int r){
    capacity=r;
}

string Car::GetColor(){
    return color;
}

int Car::GetCapacity (){
    return capacity;
}

Car::Car()//initialization using the implicit constructor (with no arguments)
{//using the scope resolution (::) operator to show belonging to class Car
    color="white";
    capacity=20;
    cout<<"Called the implicit constructor "<<endl;
}

Car::Car(string col, int cap)//initialization using constructor with 2 arguments(string, int)
{//constructors have no return type
    color=col;
    capacity=cap;
    cout<<"Called the constructor with 2 arguments "<<endl;
}

int main(){
    Car c0;//create a Car type object using the implicit constructor
    //printing it's attribute values
    cout << "Car c0 has (implicit) color "<<c0.GetColor()<<" and (implicit) tank capacity of "
         <<" "<<c0.GetCapacity()<<" liters "<<endl;

    Car c1;//create another Car type object using setters
    c1.SetColor("yellow");
    c1.SetCapacity(40);
    cout<<"Car m1 has the color "<<c1.GetColor()<<" and the tank capacity of "
        << c1.GetCapacity()<<" liters "<<endl;
```

```
    Car c2("red",35);//using the 2 argument constructor (string, int)
    cout << "Car c2 has the color "<<c2.GetColor()<<" and the tank capacity of "
         << c2.GetCapacity()<<" liters "<<endl;
    //no matter how I set the attributes, I use getters to read them for printing
    return 0;
    //destructors called for object upon exiting the scope of their usage
}
```

Homework

3.4. By using the second example as a template, create another source file where you implement the Wildlife class (or another class of your choosing) with (at least) 2 attributes, the necessary setter/getter methods as well as the implicit constructor and a constructor with parameters. Implement in main() a series of objects based on this class, and call both the implicit constructor and the constructor with parameters.