OBJECT ORIENTED PROGRAMMING

LAB 3 – CLASES AND OBJECTS (CONTINUATION), CONSTRUCTOR WITH PARAMETERS, COPY CONSTRUCTOR. FRIEND FUNCTIONS.

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. 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.

Example 1: We model a new data type a 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.
- ✓ A constructor with parameters
    - 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
    - If the object to be created has a set of arguments specified, then the compiler picks the constructor with the same number (and type) of parameters as arguments. If these 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 POLIMORPHISM 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)
- ✓ 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 check if the safety is not already set, and then set the safety ON.
- ✓ The ReleaseSafety() method will check if the safety was already released, and only after that set the safety to OFF.

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

};

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";
      }
}
```

We will use a second example in which we model the **Car** class.

This class has the **private** attributes color and tank (capacity). We create setter and getter 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

};

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
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;

}
```

In the next sections, we want to extend the Car class and add to it a copy constructor.
The copy constructor takes attribute values from an existent object and copies them over to a new object. The source and destination objects have to be of the same type (same class).
We will then create a friend function in which we compare data from the two objects (source and destination).

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

class Car
{
string color;
int capacity;

public:
void SetColor(string c);
void SetCapacity(int r);

string GetColor();
int GetCapacity();

Car(); // implicit constructor
Car(string col, int cap);  //constructor with arguments
```

```
friend bool CompareCars(const void*, const void *);  // friend function, can be taken as
a parameter

Car(const Car&);//copy constructor

};

void DecisionMessage (unsigned int*, Car*, Car*, bool (*pFctCompare)(const void *a, const
void *b));   // decision function: prototype

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

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

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

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

bool CompareCars(const void *pM1, const void *pM2)
// definition of friend function,notice... without :: (it's forbidden to use scope
resolution for friend functions)
{
return ((*(Car*)pM1).GetColor() == (*(Car*)pM2).GetColor()) ? true : false;
// need getter here. I also need the explicit pointer castinf from 'void*' to (Car*)
}

Car::Car()
// implicit construction definition
{
color="white";
capacity=20;
       cout<<"Called the constructor with 2 arguments "<<endl;
}

Car::Car(string col, int cap)
// construction with arguments definition
{
color=col;
capacity=cap;
cout<<"Called the constructor with 2 arguments "<<endl;
}

Car::Car(const Car& c1)    // copy constructor definition, argument is pointer to source
{
cout<<"copy constructor called" << endl;
color   = c1.color;
capacity = c1.capacity;
```

```
}


int main()
{
Car c0;
cout<<"Car c0 has (implicit) color "<<c0.GetColor()<<" and tank capacity of
"<<c0.GetCapacity()<<" liters "<<endl;

Car c1;
c1.SetColor("yellow");
c1.SetCapacity(40);
cout<<"Car c1 has color "<<c1.GetColor()<<" and tank capacity of "<<c1.GetCapacity()<<"
liters "<<endl;

Car c2("red",35);
cout<<"Car c2 has color "<<c2.GetColor()<<" and tank capacity of "<<c2.GetCapacity()<<"
liters "<<endl;

//copy
Car  c3 = c2;
cout<<"Car c3 has color "<<c3.GetColor()<<" and tank capacity of "<<c3.GetCapacity()<<"
liters "<<endl;

// friend function usage
unsigned int decision = 2; // specially picked to not be true/false
cout << endl;

cout << "Decision before comparing objects: " << decision << endl;
Car *pCar0 = &c0, *pCar2 = &c2;


DecisionMessage(&decision, pCar0, pCar2, CompareCars);
cout << "Decision after comparison is " <<  decision << ", meaning: " << ((decision) ?
"Same color " : "Different colors") << endl;

return 0;
}

void DecisionMessage(     unsigned int *pDecision,
Car  *pM1,
Car  *pM2,
bool (*pFctCompare)(const void *a, const void *b)
 )
{
if( 1 == pFctCompare((Car*)pM1, (Car*)pM2) )
*pDecision = true;
else
*pDecision = false;
}
```