Universidad
Carlos III de Madrid

Industrial Informatics I Lab Session

Bachelor in Industrial Electronics and Automation

2016-2017

Lab 5

Teaching staff :

Juan Carlos Gonzalez Victores

Mohamed Abderrahim

Lab 5 - Abstract Classes and Polymorphism

An **abstract class** is designed to be parent class that is inherited by child classes. Abstract classes have unimplemented methods which are inherited by, and have to be implemented in the child class, and therefore an abstract class cannot be instantiated. To be able to instantiate a child class, which inherits an abstract class, it must implement all the abstract methods.

In C++, abstract classes are declared using pure virtual member functions via the keyword `virtual`. A **pure virtual** member function is one that has to be implemented in a derived class. An abstract class is one that has at least one pure virtual member function.

```
class Operation { // abstract class
      public:
            virtual int method(int a, int b)=0;
};

class Addition: public Operation {
      public:
            int method(int a, int b) {return a+b;}
};

class Subtraction: public Operation {
      public:
            int method(int a, int b) {return a-b;}
};

int main() {
      int n1, n2;
      cout << "Introduce two integer numbers:" << endl;
      cin >> n1 >> n2;

      Addition A;
      Subtraction S;
      cout << "Adding " << n1 << "+" << n2 << "=" << A.method(n1,n2) << endl;
      cout << "Subtracting " << n1 << "-" << n2 << "=" << S.method(n1,n2) << endl;

      return 0;
}
```

As seen in the above example, a pure virtual functions is declared using the keyword **virtual**, and ending with '**=0**'.

Normally pure virtual functions are not implemented in the abstract classes, but it could be done if needed. Below is the implementation and call of a pure virtual function.

```
...

int Operation::method(int a, int b) {
      cout << "Operators " << a << " and " << b << endl;
      return 0;
}

class Addition: public Operation {
      public:
            int method(int a, int b) {
                  Operation::method(a,b);
                  return a+b;
            }
};

...
```

A pure abstract function **cannot be defined within its class**.

Declaring a member function of a base class as virtual, allows the derived classes to overwrite the virtual function. By doing so, the decision of which function needs to be called is taken not during compilation, but during runtime, and this is **dynamic linking**.

Considering the following code, and note the difference in behaviour of functions `food` and `print`. What do you think would be the output of this program? Why?

```
#include <iostream>

class Animal {
      protected:
            int age;
      public:
            Animal(){this->age=-1;}
            Animal(int n){this->age=n;}
            virtual int getAge()=0;
            virtual void print(){cout << "An animal " <<endl;}
            void food() {cout << "not defined yet" << endl;};
};

class Aquatic : public Animal {
      public:
            Aquatic():Animal(0) {}
            Aquatic(int nn):Animal(nn) {}
            int getAge() {
                  return this->age;
            }
            void print() {
                  cout << "Aquatic Animal: " << this->age << " years old" << endl;
            }
            void food() {
                  cout << "little fishes and seaweeds" << endl;
            }
};
```

```
int main() {
      Aquatic*A = new Aquatic(11);
      cout << "Aquatic:" <<endl;
      A->print();
      A->food();

      Animal *animal = a;
      cout << "Animal:" <<endl;
      animal->print();
      animal->food();

      return 0;
}
```

Polymorphism is the mechanism in which a single method identified by its name, behaves differently depending on the type of object that invokes it. That is, the ability for a method to behave differently depending on the type of object that calls it. In summary, Polymorphism is a concept that allows a number of different operations with the same name.

There are three ways of implementing polymorphism:

1. Function overloading
2. Operator overloading
3. Virtual functions

The first two cases correspond to **static polymorphism**, i.e. which function to be called is decided during compilation time. The last case corresponds to **dynamic polymorphism**, i.e. which function to be called is decided during run-time

1. Function Overloading

A common way of achieving **polymorphism** is through overloading functions or operators. Overloading is done by declare multiple functions with the same name but with different parameters list (i.e, either by differing the number or types of the parameters). The compiler will use the appropriate definition of the function, depending on the type and number of parameters used during the function call.

An example of this can be seen in the above class definition of the class `Aquatic`, where there are two constructors for this class, with different parameter list. An instance of the class `Aquatic` can be created in two different ways, with each way behaving differently:

```
...

Aquatic*A1 = new Aquatic(11);
Aquatic*A2 = new Aquatic();

...
```

Exercise 1

Modify the first example (class `Operations`) so as to be able to also perform addition and subtraction operations on three numbers by making a call to the function `method`.

2. Operator Overloading

Operator overloading provides additional functionality to C++ operators such as +, *, >, =, +, =, etc., when applied to user defined data types.

Continuing with the class `Animals` example, the following code illustrates operator overloading for both unary (++) and binary (>) operators:

```
...

class Aquatic: public Animal {
...
    void operator ++(){this->age++;} //prefix
    void operator ++(int){this->age++;} //postfix
    bool operator>(Aquatic A) {return this->age > A.age;}
};

...

int main(){
      ...
      Aquatic A1(1);
      Aquatic A2(2);
      cout << "A1 age:" << A1.getAge() << endl;
      cout << "A2 age:" << A2.getAge() << endl;

      if(A1> A2) {
            cout << "A1>A2" << endl;
      }
      else {
            cout << "A1<=A2" << endl;
      }

      ++A1;
      A1++;

      cout << "A1 age:" << A1.getAge() << endl;
      cout << "A2 age:" << A2.getAge() << endl;

      if(A1>A2) {
            cout << "A1>A2 " << endl;
      }
      else {
            cout << "A1<=A2 " << endl;
      }
      ...
}
```

However, not all operators can be overloaded. The following is the list of operators that cannot be overloaded:

Operator definition	Operator
Member access	`.` (dot operator)
Scope resolution	`::` (global access)
Conditional	`?:` (conditional statement)
Pointer to member	`* (asterisk)`
Size of data type	`sizeof(...)`

Exercise 2

Implements operator overloading for `>>` in class `Animals`, such that it allows assigning a value to the member age of the object.

3. Virtual Functions

Defining an abstract class by declaring a virtual function in it, allows dynamic polymorphism as we shall see. The following example illustrates this, where in, which operation to be performed is determined during runtime.

```
class Operation {
      virtual int method(int a, int b)=0;
};

class Addition: Operation {
      public:
            int method(int a, int b) {return a+b;}
};

class Subtraction: Operation {
      public:
            int method(int a, int b) {return a-b;}
};

class Multiplication: public Operation {
      public:
            int method(int a, int b) {return a*b;}
};
```

```
int main() {
	int n1, n2;
	cout << "Introduce dos enteros:" <<endl;
	cin >> n1 >> n2;

	srand(time(NULL));	/* initialize random seed: */
	Operation *O;
	switch(rand() % 3){
		case 0: O = new Addition;
			cout << "Operation ADDITION" <<endl;
			break;
		case 1: O = new Subtraction;
			cout << "Operation SUBTRACTION" <<endl;
			break;
		case 2:O = new Multiplication;
			cout << "Operation MULTIPLICATION" <<endl;
			break;
	}
	cout << "Result: " <<O->method(n1, n2) <<endl;
}
```

Exercise

- You need to extend the Computer Simulator project that was started in the previous lab session. Based on what you have already implemented, you need to make the necessary changes to the project to implement the new extended UML diagram, as provided in annex. Descriptions of the classes and their methods are also provided in the annex.
- The following should be the main function of the program:

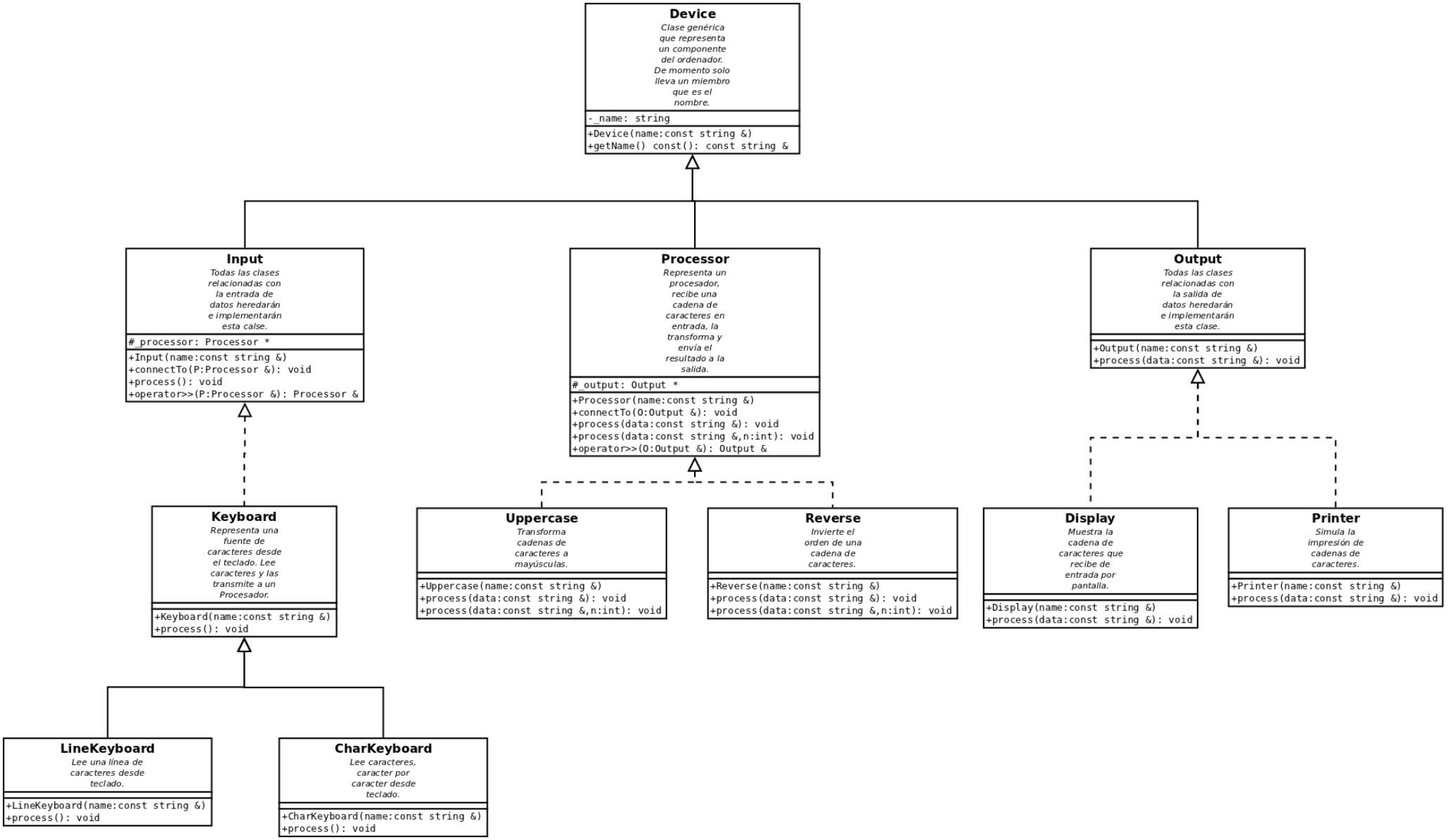
```
int main() {
      LineKeyboard LK("logitech");
      Uppercase UP("intel");
      Display D("lg");
      Input *I = &LK;
      LK.connectTo(UP);
      I->connectTo(UP); // do same as before line
      UP.connectTo(D);
      LK.process();

      CharKeyboard CK("logitech char");
      Reverse R("intel reverse");
      Printer P("Ricoh");
      I = &CK;
      (*I) >> R>>P;
      I->process();
}
```

Annex

Following is a description of all the classes and their methods:

Class	Function	Description
`Device`		Remains the same as before.
`Input`	`Input::Input(const string & name)`	Initialize an input device with the specified name.
	`Input::connectTo(Processor & P)`	Connect the input device with a processor.
	`Processor Input::operator>>(Processor & P)`	Connect the input device with a processor.
	`void Input::process()`	Pure virtual function.
`Keyboard`	`Keyboard::Keyboard(const string & name)`	Initialize a keyboard with the specified name.
	`void Keyboard::process()`	Default implementation of a keyboard, reads a string of real characters from the keyboard using the flow operator `cin`. Then processes this string by calling the `process()` method of the processor object connected to this keyboard object.
`CharKeyboard`	`CharKeyboard::CharKeyboard(const string & name)`	Initialize a character keyboard with the specified name.
	`void CharKeyboard::process()`	This is similar to the `Keyboard::Process()` function, but it only reads a single character (To do this you can use the standard function `cin.get()`).
`LineKeyboard`	`LineKeyboard::LineKeyboard(const string & name)`	Initialize a line keyboard with the specified name.
	`void LineKeyboard::process()`	This is similar to the `Keyboard::process()` function, but reads a complete line, i.e, reads characters until `'\n'`, and send a full line to the processor.
`Processor`	`Processor::Processor(const string & name)`	Initialize a processor with the specified name.
	`void Processor::connectTo(Output  & O)`	Connect the `Processor` object to a monitor screen (`Display`) object.
	`void Processor::process(const string & data)`	Virtual member function that is going to search for special commands in a given input string, such as the command "`quit`".
	`void Processor::process(const string & data, int n)`	The same as the above function, but this function will search for the special command in the first *n* characters.

	`Output Processor::operator>>(Output & O)`	Connect the `Processor` object to a monitor screen (`Display`) object.
`Uppercase`	`Uppercase::Uppercase(const string & name)`	Initialize an uppercase processor with the specified name.
	`void Uppercase::process(const string & data)`	Converts all the character in the input string to uppercase, adds the prefix `"PROCESSED:"`, and then sends the processed string to the connected monitor screen (`Display`) object. For example, if the input string is `"hello"`, then the processed data string sent to the monitor screen would be `"PROCESSED: HELLO"`.
	`void Uppercase::process(const string & data, int n)`	The same as the above function, but this function considers only the first *n* characters.
`Reverse`	`Reverse::Reverse(const string & name)`	Initialize a *'reverse'* processor with the specified name.
	`void Reverse::process(const string & data)`	Inverts the character string it receives as input, adds the prefix `"PROCESSED:"`, and then sends the processed string to the connected monitor screen (`Display`) object. For example, if the input string is `"hello"`, then the processed data string to be sent to the monitor screen is `"PROCESSED: olleh"`.
	`void Reverse::process(const string & data, int n)`	The same as the above function, but this function considers only the first *n* characters.
`Output`	`Output::Output(const string & name)`	Initialize an output with the specified name.
	`void Output::process()`	Pure virtual function.
`Display`	`Display::Display(const string & name)`	Initialize a monitor screen with the specified name.
	`void Display::process(const string & data)`	Displays the input data string on the screen using `cout`.
`Printer`	`Printer::Printer(const string & name)`	Initialize a printer with the specified name.
	`void Printer::process(const string & data)`	Simulate printing by displaying the received input string prefixing it with **"`Printing...`"**. For example, if the input string is **"`PROCESSED: olleh`"**, then print on the screen using `cout` the string **"`Printing... PROCESSED: olleh`"**.


Device
Clase genérica que representa un componente del ordenador. De momento solo lleva un miembro que es el nombre.
-_name: string
+Device(name:const string &)
+getName() const(): const string &
Input
Todas las clases relacionadas con la entrada de datos heredarán e implementarán esta calse.
processor: Processor *
+Input(name:const string &)
+connectTo(P:Processor &): void
+process(): void
+operator>>(P:Processor &): Processor &
Processor
Representa un procesador, recibe una cadena de caracteres en entrada, la transforma y envía el resultado a la salida.
output: Output *
+Processor(name:const string &)
+connectTo(O:Output &): void
+process(data:const string &): void
+process(data:const string &,n:int): void
+operator>>(O:Output &): Output &
Output
Todas las clases relacionadas con la salida de datos heredarán e implementarán esta clase.
+Output(name:const string &)
+process(data:const string &): void
Keyboard
Representa una fuente de caracteres desde el teclado. Lee caracteres y las transmite a un Procesador.
+Keyboard(name:const string &)
+process(): void
Uppercase
Transforma cadenas de caracteres a mayúsculas.
+Uppercase(name:const string &)
+process(data:const string &): void
+process(data:const string &,n:int): void
Reverse
Invierte el orden de una cadena de caracteres.
+Reverse(name:const string &)
+process(data:const string &): void
+process(data:const string &,n:int): void
Display
Muestra la cadena de caracteres que recibe de entrada por pantalla.
+Display(name:const string &)
+process(data:const string &): void
Printer
Simula la impresión de cadenas de caracteres.
+Printer(name:const string &)
+process(data:const string &): void
LineKeyboard
Lee una línea de caracteres desde teclado.
+LineKeyboard(name:const string &)
+process(): void
CharKeyboard
Lee caracteres, caracter por caracter desde teclado.
+CharKeyboard(name:const string &)
+process(): void