SOMETING SOMETHING OBJECT ORIENTED

NANOTEKSPICE BOOTSTRAP

SOMETING SOMETHING OBJECT ORIENTED

Electronic Simulator

The **NanoTekSpice** project being a digital electronic simulator based on a graph, the first thing you need to do is to discover what digital electronic is...

Here is an online digital electronic simulator. It lets you add components, link them together and run simulations. What you are trying to recreate is similar to this, with some additional real components.

There are three states in digital electronics: `True`, `False` and `Undefined`. On **logic.ly**, `True` makes wires appear in blue, `Undefined` makes them appear in grey and `False` in white.

`Undefined` is the default state of every part of a circuit. Input states may make `Undefined` states disappear if the logic if the circuit allows it

In the end, this is all very similar to normal variables. Here are a few examples:

```
{
        bool a = true;
        bool b;
        bool c;

        c = a && b;
        // c is undefined because b is undefined and a is true, so the result of c depends
             only on b
}

{
        bool a = false;
        bool b;
        bool c;

        c = a && b;
        // c is false, because false AND anything is always false
}

{
        bool a = true;
        bool b;
        bool c;

        c = a || b;
        // c is true because true OR anything is always true
}
```

```
{
        bool a = false;
        bool b;
        bool c;

        c = a || b;
        // c is undefined because b is undefined and a is false, so the result of c
            depends only on b
}
```

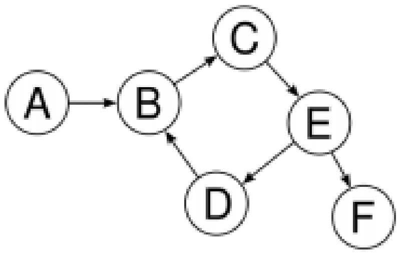
Truth tables for AND, OR and XOR gates can be found in the project subject.

Create a graph

A graph is a data structure in which nodes are linked together, without any specific shape.

- ✓ A tree has a specific shape: there is always a root node with children, each of these children being the root of other children...
- ✓ A linked list has a specific shape: there is one node, linked to another, which may be linked again, and so on and so forth.

A graph's only requirement is that nodes are linked together. Therefore, trees and lists are two subsets of graphs, just like lists are a subset of trees.



Graphs can be oriented, meaning that links are unidirectional. In the example above, A is linked to B but B is not linked to A. Note that nodes can also be linked to themselves.

In the **NanoTekSpice**, components are the nodes, and the circuit hold the whole graph. The `Circuit` class contains the components, manipulating them through their `IComponent` interface.

As every components need to have the means of linking itself to the other components, why not create an intermediary class implementing the link management for everyone ? It would only leave the component logic to implement in our final components. Also, factoring your code is a brillant idea : it avoids the duplication of our code, making it more maintainable.

Declare and implement the `nts::AComponent` class which:

- ✓ Represents a component with inputs and outputs
- ✓ Inherit from `nts::IComponent`
- ✓ Leave the `compute` method pure virtual
- ✓ Implement an empty `simulate` method
- ✓ Can be linked to other `IComponents` using the `setLink` method
- ✓ Add a `getLink(std::size_t pin)const` method calling `compute` on the linked component.

`nts::IComponent::simulate` is only useful for `clock`, `input` and advanced components.

You can now easily create the basic components inheriting from `AComponent`, you just have to implement the `compute` method (and `simulate` for some of them). It's time to implement the basic components of the project :

- ✓ **Special components** : `nts::InputComponent`, `nts::OutputComponent`, `nts::TrueComponent`, `nts::FalseComponent`, `nts::ClockComponent`
- ✓ **Elementary components** : `nts::AndComponent`, `nts::OrComponent`, `nts::XorComponent`, `nts::NotComponent`

The following code should compile and produce the following result :

```cpp
std::ostream&   operator<<(std::ostream& s, nts::Tristate v)
{
    /* Implement me */
}

int main(void)
{
    std::unique_ptr<nts::IComponent> gate = std::make_unique<nts::AndComponent>();
    std::unique_ptr<nts::IComponent> input1 = std::make_unique<nts::FalseComponent>();
    std::unique_ptr<nts::IComponent> input2 = std::make_unique<nts::TrueComponent>();
    std::unique_ptr<nts::IComponent> inverter = std::make_unique<nts::NotComponent>();

    gate->setLink(1, *input1, 1);
    gate->setLink(2, *input2, 1);
    inverter->setLink(1, *gate, 3);
    std::cout << "!(" << input1->compute(1) << " && " << input2->compute(1) << ") -> " <<
        inverter->compute(2) << std::endl;
}
```

Terminal

```
~/B-OOP-400> ./a.out
!(0 && 1) -> 1
```

You need a class to store the component graph. Declare and implement the `Circuit` class :

- ✓ Manipulate `nts::IComponents`
- ✓ Add a component with its name
- ✓ Find a component by its name
- ✓ Simulate a tick for each components
- ✓ Display the state of its input and output components

Take all the time you need to design your architecture. Read the subject multiple times. You should write your declarations (`.hpp`) before your implementations (`.cpp`).

A `Circuit` has inputs and outputs, just like... a `nts::IComponent` ?

v 2.3

{EPITECH}