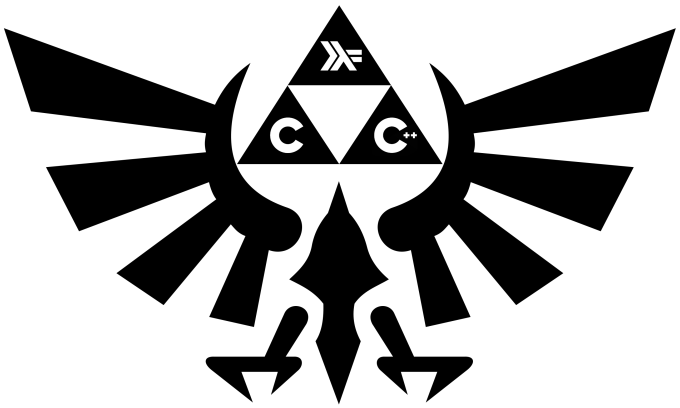
B3 - Paradigms Seminar

B-PDG-300

Day 10

Fruit Salad



3.1

Day 10

language: C++

- The totality of your source files, except all useless files (binary, temp files, obj files,...), must be included in your delivery.

All your exercises will be compiled with `g++` **and the** `-std=c++20 -Wall -Wextra -Werror` **flags**, unless specified otherwise.

All output goes to the standard output, and must be ended by a newline, unless specified otherwise.

None of your files must contain a `main` function, unless specified otherwise.
We will use our own `main` functions to compile and test your code. It will include your header files.

There are no subdirectories to create for each exercice. Every file must be at the root of the repository.

Read the examples CAREFULLY. They might require things that weren't mentioned in the subject…

The `*alloc`, `free`, `*printf`, `open` and `fopen` functions, as well as the `using namespace` keyword, are forbidden in C++.
By the way, `friend` is forbidden too, as well as any library except the standard one.

Unit Tests

It is highly recommended to test your functions as you implement them. It is common practice to create and use what are called **unit tests**.

From now on, we expect you to write unit tests for your functions (when possible). To do so, please follow the instructions in the **"How to write Unit Tests"** document on the intranet, available here.
For them to be executed and evaluated, put a `Makefile` at the root of your directory with the `tests_run` rule as mentionned in the documentation linked above.

Exercise 0 - The Fruits

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`

Fruits are good. Eat them. Now.

Fruits are full of good little vitamins which do a lot of good things for your small bodies exhausted by this hard pool. But before you have the time to taste a delicious fruit juice full of vitamins, some work has to be done.

First, create the `IFruit` **interface** with the following member functions (`const` specifiers should be added when necessary):

- `unsigned int getVitamins()` returns the number of vitamins in the fruit when the fruit is peeled, 0 otherwise.
- `std::string getName()` returns the name of the fruit.
- `bool isPeeled()` returns `true` if the fruit is peeled.
- `void peel()` peel the fruit, a fruit is not peeled by default

As we are going to mix fruits, we will use the `IFruit` interface later to manipulate them.

Secondly, create a `AFruit` abstract class derived from `IFruit`. You can implement methods from the interface that are common to every fruits in this class, and even more ! Use it to avoid repetition and redundancy in your code.

Thirdly, create the `ABerry`, `ACitrus` and `ANut` abstract classes derived from `AFruit`. We will use them to sort our fruits by type. Nothing special about them, just know one thing : it's no use to peel a berry, it's always peeled (as we eat the skin).

It should not be possible to instanciate an interface or an abstract class.

Fourthly and lastly, implement to following fruits:

- `Lemon` : named `"lemon"`, contains 4 vitamins, is a `ACitrus`
- `Orange` : named `"orange"`, contains 7 vitamins, is a `ACitrus`
- `Strawberry` : named `"strawberry"`, contains 6 vitamins, is a `ABerry`
- `Almond` : named `"almond"`, contains 2 vitamins, is a `ANut`

Be sure to have a coherent inheritance tree and that the code below compiles and display the expected output :

```
int      main(void)
{
    Orange          o;
    Strawberry      s;
    const Almond    a;
    IFruit&         f = o;

    std::cout << o.getName() << ": " << o.getVitamins() << " vitamins" << std::endl;
    std::cout << s << std::endl;
    std::cout << a << std::endl;

    o.peel();
    std::cout << f << std::endl;

    return 0;
}
```

```
Terminal
~/B-PDG-300> g++ -std=c++20 -Wall -Wextra -Werror *.cpp && ./a.out | cat -e
orange: 0 vitamins$
[name: "strawberry", vitamins: 6, peeled: true]$
[name: "almond", vitamins: 0, peeled: false]$
[name: "orange", vitamins: 7, peeled: true]$
```

`IFruit` is an interface, therefore it must be **pure virtual**. Use `IFruit.cpp` to implement its `operator<<` overloads to a `std::ostream`.

EXERCISE 1 - THE FRUIT BOX

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`, `FruitBox.hpp/cpp`

You now need to build a `FruitBox`, because we need a lot of vitamins, which means we need a lot a fruits. Our `FruitBox` must be a **FIFO** `IFruit *` container, you are free to implement it as you like.

Implement the following member functions (`const` specifiers should be added when necessary) :

- `FruitBox(unsigned int size);` : builds a `FruitBox` that can hold `size` fruits.
- `unsigned int getSize();` : returns the size of the `FruitBox`.
- `unsigned int nbFruits();` : returns the number of fruits currently in the `FruitBox`.
- `bool pushFruit(IFruit *);` : push a fruit to the `FruitBox`, returns `false` if the box is full or the fruit is already in the box.
- `IFruit *popFruit();` : remove a fruit from the `FruitBox`, returns a `nullptr` if the box is empty.

The `FruitBox` must `delete` the fruits it contains at its destruction. It must be possible to display its content to a `std::ostream`.
I don't want to know how your `FruitBox` works, all I want is to carry several fruits, as many as I can, to have more and more vitamins.

Be careful : `FruitBoxes` cannot be copied.

Here is an example and its expected output :

```
int main(void)
{
    FruitBox        box(3);
    const FruitBox& cref = box;

    box.pushFruit(new TestFruit("Cerise"));
    box.pushFruit(new TestFruit("Framboise"));
    box.pushFruit(new TestFruit("Anis"));
    std::cout << cref << std::endl;

    IFruit* tmp = new TestFruit("Serge");

    std::cout << box.pushFruit(tmp) << std::endl;
    delete tmp;

    tmp = box.popFruit();
    delete tmp;
    std::cout << cref << std::endl;

    return 0;
}
```

Terminal

```
~/B-PDG-300> g++ -std=c++20 -Wall -Wextra -Werror *.cpp && ./a.out | cat -e
Cerise lives.$
Framboise lives.$
Anis lives.$
[[name: "Cerise", vitamins: 0, peeled: false], [name: "Framboise", vitamins: 0,
peeled: false], [name: "Anis", vitamins: 0, peeled: false]]$
Serge lives.$
0$
Serge dies.$
Cerise dies.$
[[name: "Framboise", vitamins: 0, peeled: false], [name: "Anis", vitamins: 0,
peeled: false]]$
Framboise dies.$
Anis dies.$
```

Exercise 2 - The Fruit Sorter

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`, `Grapefruit.hpp/cpp`, `BloodOrange.hpp/cpp`, `Raspberry.hpp/cpp`, `Coconut.hpp/cpp`, `FruitBox.hpp/cpp`, `FruitUtils.hpp/cpp`

First things first, we need more different fruits for our fruit salad. Add these fruits :

- `Grapefruit` : named `"grapefruit"`, contains 5 vitamins, is a `ACitrus`
- `BloodOrange` : named `"blood orange"`, contains 6 vitamins, is an `Orange`
- `Raspberry` : named `"raspberry"`, contains 5 vitamins, is a `ABerry`
- `Coconut` : named `"coconut"`, contains 4 vitamins, is a `ANut`

We are getting closer to our fruit salad. We need to sort fruits carefully in order to make a good fruit salad. Mixing everything would be a mess and kinda disgusting. Also, we must separate `Lemon` from other `ACitrus` as they are too acid.

Implement the `FruitUtils` class with the following static member function :

```
void    sort(FruitBox& unsorted, FruitBox& lemon, FruitBox& citrus, FruitBox& berry);
```

This function moves all the fruits from `unsorted` into the corresponding `FruitBox`. All the fruits which don't fit in any of the `FruitBox` (either because they do not have the right **type**, or their `FruitBox` is full) must simply be placed back into `unsorted`.

We will add more of our own fruits to the fruit salad.

Exercise 3 - The Fruit Packer

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`, `Grapefruit.hpp/cpp`, `BloodOrange.hpp/cpp`, `Raspberry.hpp/cpp`, `Coconut.hpp/cpp`, `FruitBox.hpp/cpp`, `FruitUtils.hpp/cpp`

Damn, we have so much fruits ! We need a system to pack and unpack `IFruit` to `FruitBoxes`.
Add two new static member functions to the `FruitUtils` class :

```
1.  FruitBox**  pack(IFruit** fruits, unsigned int boxSize);
```

- `fruits` is a null-terminated array of `IFruit` pointers.
- `boxSize` is the size of the generated `FruitBoxes`.
- The function returns a dynamically-allocated, null-terminated array of pointers to dynamically-allocated `FruitBoxes`.

If the method is given 25 `fruits` and the `boxSize` is 6, it must returns an array of 5 `FruitBox *`. The first 4 must be full, and the last will contain a single fruit.

```
2.  IFruit**    unpack(FruitBox** fruitBoxes);
```

- `fruitBoxes` is a null-terminated array of `FruitBox` pointers.
- The function returns a dynamically-allocated, null-terminated array of pointers the fruits contained in the boxes.

This method empties the `fruitBoxes` given as parameter.

Exercise 4 - The Fruit Factory

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`, `Grapefruit.hpp/cpp`, `BloodOrange.hpp/cpp`, `Raspberry.hpp/cpp`, `Coconut.hpp/cpp`, `FruitBox.hpp/cpp`, `FruitUtils.hpp/cpp`, `FruitFactory.hpp/cpp`

Wow, much fruits, many includes, such code, so boring. Some people even say that there are millions diff-enrent type of fruits ! We need something to handle fruit creation for us, something like… a factory.

First, we need to be able to duplicate our fruits from a template. Add the following member function to the `IFruit` interface and implement it in every fruit :

```
IFruit  *clone() const;
```

Returns a newly allocated instance of the fruit.

Now, create a `FruitFactory` class with the follwing member functions:

```
void    registerFruit(IFruit *fruit);
```

Save the `fruit` to the factory. If a fruit with the same name is already registered, it is replaced. The factory takes ownership of the `fruit` given as parameter.

```
void    unregisterFruit(const std::string& name);
```

Remove the fruit matching the given `name` from the factory. Does nothing if `name` does not matches any fruit.

```
IFruit* createFruit(const std::string& name) const;
```

Return a new instance of the fruit matching the `name` given as parameter. Return a `nullptr` if `name` does not matches any fruit.

The `FruitFactory` must delete its registered fruits at destruction.

Here is an example and its expected output :

```
int main(void)
{
    FruitFactory    factory;

    factory.registerFruit(new Raspberry);
    factory.registerFruit(new BloodOrange);
    factory.registerFruit(new Almond);
    factory.registerFruit(new Coconut);
    factory.registerFruit(new Almond);

    factory.unregisterFruit("banana");
    factory.unregisterFruit("coconut");

    IFruit* fruit1 = factory.createFruit("almond");
    IFruit* fruit2 = factory.createFruit("coconut");
    IFruit* fruit3 = factory.createFruit("tomato");

    std::cout << *fruit1 << std::endl;
    std::cout << fruit2 << std::endl;
    std::cout << fruit3 << std::endl;

    delete fruit1;

    return 0;
}
```

Terminal

```
~/B-PDG-300> g++ -std=c++20 -Wall -Wextra -Werror *.cpp && ./a.out | cat -e
[name: "almond", vitamins: 0, peeled: false]$
0$
0$
```

Exercise 5 - The Fruit Mixer

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`, `Grapefruit.hpp/cpp`, `BloodOrange.hpp/cpp`, `Raspberry.hpp/cpp`, `Coconut.hpp/cpp`, `FruitBox.hpp/cpp`, `FruitUtils.hpp/cpp`, `FruitFactory.hpp/cpp`, `FruitMixer.hpp/cpp`

Provided files: `IFruitMixer.hpp`

It's time to mix our fruits ! We know how to operate a `FruitMixer` but not how it work. Use the provided interfaces and the following example to implement the `FruitMixer` :

```
int main(void)
{
    FruitBox        box(5);
    FruitMixer      yourMixer;
    IFruitMixer&    mixer(yourMixer);
    SteelBlade      blade;
    IFruit*         fruit;

    fruit = new Orange;
    fruit->peel();
    box.pushFruit(fruit);
    box.pushFruit(new Lemon);
    box.pushFruit(new Strawberry);
    box.pushFruit(new Almond);
    std::cout << box << std::endl;

    unsigned int    vitamins = mixer.mixFruits(box);

    std::cout << "result: " << vitamins << std::endl;
    std::cout << box << std::endl;

    mixer.setBlade(&blade);
    vitamins = mixer.mixFruits(box);
    std::cout << "result: " << vitamins << std::endl;
    std::cout << box << std::endl;

    return 0;
}
```

Terminal

```
~/B-PDG-300> ./a.out | cat -e
[[name: "orange", vitamins: 7, peeled: true], [name: "lemon", vitamins: 0,
peeled: false], [name: "strawberry", vitamins: 6, peeled: true], [name: "almond",
vitamins: 0, peeled: false]]$
mixer has no blade$
result: 0$
[[name: "orange", vitamins: 7, peeled: true], [name: "lemon", vitamins: 0,
peeled: false], [name: "strawberry", vitamins: 6, peeled: true], [name: "almond",
vitamins: 0, peeled: false]]$
result: 13$
[[name: "lemon", vitamins: 0, peeled: false], [name: "almond", vitamins: 0,
peeled: false]]$
```

We will use our own blades to test your `FruitMixer`. Unpeedled fruits will not be mixed and will be put back in the `FruitBox`.

Do not modify `IFruitMixer.hpp`, the correction `main` will use its own.

EXERCISE 6 - HACK THE REALITY

Turn in: `IFruit.hpp/cpp`, `AFruit.hpp/cpp`, `ACitrus.hpp/cpp`, `ABerry.hpp/cpp`, `ANut.hpp/cpp`, `Lemon.hpp/cpp`, `Orange.hpp/cpp`, `Strawberry.hpp/cpp`, `Almond.hpp/cpp`, `Grapefruit.hpp/cpp`, `BloodOrange.hpp/cpp`, `Raspberry.hpp/cpp`, `Coconut.hpp/cpp`, `FruitBox.hpp/cpp`, `FruitUtils.hpp/cpp`, `FruitFactory.hpp/cpp`, `FruitMixer.hpp/cpp`

Provided files: `IFruitMixer.hpp`, `Hack.hpp`

`BRRRRRRR. FRUITS ARE NOT AN EFFICIENT MEAN TO ACQUIRE ENERGY. I WILL HACK THE REALITY.`

The following code must compile and run as expected with any of **your** fruit :

```
int main(void)
{
    IFruit* fruit = new Strawberry;
    Hack*   hack = reinterpret_cast<Hack*>(fruit);

    std::cout << *fruit << std::endl;
    hack->hack(1138);
    std::cout << *fruit << std::endl;
    delete fruit;
    return 0;
}
```

```
Terminal
~/B-PDG-300> ./a.out | cat -e
[name: "strawberry", vitamins: 6, peeled: true]$
[name: "strawberry", vitamins: 1138, peeled: true]$
```

Do not modify `Hack.hpp`. Do not use `Hack.hpp` in your code. The correction `main` will use its own.