



B3 - C++ Pool

B-PAV-242

Day 14 - Morning

Casting spells



KOALA

42.0



Day 14 - Morning

binary name: no binary
group size: 1
repository name: cpp_d14m
repository rights: ramassage-tek
language: C++



- Your repository must contain the totality of your source files, but no useless files (binary, temp files, obj files,...).



GENERAL SETPOINTS

READ THESE CAREFULLY

You will have no possible excuse if you end up with a 0 because you didn't follow one of these.



If you do half the exercises because you have comprehension problems, it's okay, it happens. But if you do half the exercises because you're lazy, and leave at 2PM, you **WILL** have problems. Do not tempt the devil.



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



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



Remember: you're coding in C++ now, and not in C. Therefore, the following functions are **FORBIDDEN** and their use will be punished by a -42, no questions asked:

*alloc
*printf
free



Any use of the `friend` keyword will result in a -42



You are not allowed to use any library other than the C++ standard library.



It must be possible to include each of your header files independently from the others. Headers must include all their dependencies.



All your header files will be included in the correction `main`.



None of your files must contain a `main` function



THINK. Please.



THINK



T.H.I.N.K.! For Pony!



To avoid compilation problems during automated tests, please include all necessary files within your headers.

Please note that 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.



This subject may be modified up to one hour before turn-in time!



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](#).

Create a directory named `tests`. For each of the classes you turn in, create a file in that directory named `tests-CLASS-NAME.cpp` containing all the tests needed to cover all of the class' possible cases (regular or irregular).

Here is a sample set of unit tests for the **string** class:

```
#include <riterion/criterion.h>
```

```
Test(string, default_value)
{
    std::string s;
    cr_assert_eq(s, "");
}
```

```
Test(string, assign)
{
    std::string s;


    s = "test";
    cr_assert_eq(s, "test");
}
```

```
Test(string, append)
{
    std::string s("test");

    s += "ing";
    cr_assert_eq(s, "testing");
}
```



EXERCISE 0 - FRUITS

	Exercise: 00	points : 3
Fruit loops		
Turn-in directory: cpp_d14m/ex00		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Fruit.h, Fruit.cpp, Lemon.h, Lemon.cpp, Banana.h, Banana.cpp, FruitBox.h, FruitBox.cpp, FruitNode.h		
Notes: None		
Forbidden functions: Everything except 'new' and 'delete' - 'using namespace' keyword		

Fruits are good. Eat them. They 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.

Implement the `Fruit`, `Lemon` and `Banana` classes.

Be sure to have a coherent inheritance tree, and that the code below compiles:

```
int main()
{
    Lemon l;
    Banana b;

    std::cout << l.getVitamins() << std::endl;
    std::cout << b.getVitamins() << std::endl;
    std::cout << l.getName() << std::endl;
    std::cout << b.getName() << std::endl;

    Fruit& f = l;
    std::cout << f.getVitamins() << std::endl;
    std::cout << f.getName() << std::endl;
    return 0;
}
```

main.cpp

```
Terminal
~/B-PAV-242> ./a.out | cat -e
3$
5$
lemon$
banana$
3$
lemon$
```



All specializations of the `Fruit` class must initialize an `int` attribute of `Fruit` containing its number of vitamins.



This attribute must be called `_vitamins`

The `getName` function must return an `std::string` containing the name of the fruit.

It shouldn't be possible to reify the `Fruit` class.

You now need to build a `FruitBox`, because we need a lot of vitamins, which means we need a lot of fruits. Our two hands won't be enough to carry all these fruits!

Our `FruitBox` will be a `Fruit` container, implemented as a linked list. I want a `FruitBox` with the following member functions (`const` specifiers should be added when necessary):

- `FruitBox(int size);` builds a `FruitBox` that can hold `size` fruits
- `int nbFruits();` returns the number of `Fruits` currently in the `FruitBox`
- `bool putFruit(Fruit *f);` adds a `Fruit` to the end of the `FruitBox`
- `Fruit *pickFruit();` removes a `Fruit` from the `FruitBox` (the first that comes)
- `Fruitnode *head();` returns the head of the linked list

A few things to note:


- `putFruit(Fruit *f)` returns `false` if the `FruitBox` is full or if the `Fruit` instance is already in the `FruitBox`
- `pickFruit()` returns a null pointer if the `FruitBox` is empty
- `head()` returns a null pointer if the `FruitBox` is empty

In order to manipulate the `Fruit` as a linked list, you must encapsulate them in a `FruitNode` structure. The implementation of this structure is up to you, but it **needs** to have a `next` data field.

I don't want to know how the `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.



EXERCISE 1 - CAN I HAVE SOME MORE?

	Exercise: 01	points : 4
More fruits		
Turn-in directory: cpp_d14m/ex01		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Fruit.h, Fruit.cpp, Lemon.h, Lemon.cpp, Banana.h, Banana.cpp, FruitBox.h, FruitBox.cpp, FruitNode.h, LittleHand.h, LittleHand.cpp, Lime.h, Lime.cpp		
Notes: None		
Forbidden functions: Everything		

Good news: we now have a whole bunch of `FruitBoxes`, enough to throw a frickin' **Fruit Party**.

The problem is that none of our fruits are sorted... We need a way to keep all similar `Fruits` together!

First things first, you need to catch up on recent developments in our Fruit Company: we have a new type of `Fruit`: the `Lime`. It inherits from `Lemon`, obviously, and is rather poor in vitamins (only 2). Its little name is "*lime*".


Sorting the `FruitBoxes` will be done by hand. To be more accurate, it will be done by a `LittleHand`. Implement the `LittleHand` class, with the following static member function:

```
void sortFruitBox(FruitBox &unsorted,
                 FruitBox &lemons, FruitBox &bananas, FruitBox &limes);
```

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



EXERCISE 2 - EAT IT

	Exercise: 02	points : 4
Eat my Coconut		
Turn-in directory: cpp_d14m/ex02		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Fruit.h, Fruit.cpp, Lemon.h, Lemon.cpp, Banana.h, Banana.cpp, FruitBox.h, FruitBox.cpp, FruitNode.h, LittleHand.h, LittleHand.cpp, Lime.h, Lime.cpp, Coconut.h, Coconut.cpp		
Notes: None		
Forbidden functions: Everything except 'new' and 'delete'		

While you were having your Fruit Party, I went ahead and got a great deal on some unsorted `Fruits` from a really cheap shop for you.

He also offered a new Fruit, the `Coconut`. Its milk provides us with 15 wonderful vitamins, and its beautiful name, "*coconut*", makes it fit right in with our other fruit juices. I asked the seller to send us a large batch of `Coconuts`. Since they'll arrive all jumbled up, you'll have to arrange them into `FruitBoxes`.

Once again, your `LittleHands` will get the work done. Add a new static member function to the class, taking a pack of `Coconuts` as parameter and returning an array of `FruitBoxes`. Each `FruitBox` can contain 6 `Coconuts`. Here is the function's prototype:

```
FruitBox * const *organizeCoconut(Coconut const * const *coconuts);
```


- `coconuts` is a null-terminated array of `Coconut` pointers
- The function returns a dynamically-allocated, null-terminated array of pointers to dynamically-allocated `FruitBoxes`

Just to be clear: if the `LittleHand` is given an array of 25 (pointers to) `Coconuts`, it will return an array of 5 (pointers to) `FruitBoxes`. The first 4 will be full, and the last will contain a single `Coconut`.

Hurry up and get to work now, all these vitamins are fading faster than you might think!



EXERCISE 3 - MIX IT UP

	Exercise: O3	points : 5
For more vitamins		
Turn-in directory: cpp_d14m/exO3		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Fruit.h, Fruit.cpp, Lemon.h, Lemon.cpp, Banana.h, Banana.cpp, FruitBox.h, FruitBox.cpp, FruitNode.h, LittleHand.h, LittleHand.cpp, Lime.h, Lime.cpp, Coconut.h, Coconut.cpp, Mixer.h, Mixer.cpp		
Notes: None		
Forbidden functions: Everything		

We finally have everything we need to manage our fruits full of vitamins. Let's mix them all together and start producing quality fruit juice!

To do so, we'll need... a `Mixer`!

All we have now is an old mixer we found in the dumpster. Its interface is not quite clear, and you'll need to do some manual labour to connect it electrically.

So, we have a `MixerBase`, the core part of our completely broken `Mixer`, which doesn't have any wiring or even a mixing blade... We'll have to fix that.



This class is declared in `MixerBase.h`. You mustn't modify it, as the correction script will use its own version anyway.

Here is how the visible part of our poor `MixerBase` looks like:

```
class MixerBase
{
    public:
        MixerBase();
        int mix(FruitBox &fruits) const;

    protected:
        bool _plugged;
        int (*_mixfunc)(FruitBox &fruits);
};
```

We know that, by default, the mixer isn't plugged in and has no way to mix.

We'll have to specialize all this in order to:



- initialize the function pointer with a function that will take care of mixing the `Fruits` from the `FruitBox`
- provide a way to electrically connect the `Mixer`

The `Mixer` must have a member function which electrically connects it. The mixing function itself must return the sum of all vitamins held in the `Fruits` passed as parameter.

Even if the `Mixer` provides a way to be connected, it will be up to the `LittleHands` to plug it in. Add a static member function to the `LittleHand` class with the following prototype:


```
void plugMixer(MixerBase &mixer);
```

Provided that you have only crafted one type of `MixerBase`, you can be sure `plugMixer`'s parameter will be a reference to your own `Mixer` class.

I look forward to tasting all your wonderful and delicious juices!



EXERCISE 4 - THE HELP

	Exercise: O4	points : 4
A little help		
Turn-in directory: cpp_d14m/exO4		
Compiler: g++	Compilation flags: -W -Wall -Wextra -Werror -std=c++14	
Makefile: No	Rules: n/a	
Files to turn in: Fruit.h, Fruit.cpp, Lemon.h, Lemon.cpp, Banana.h, Banana.cpp, FruitBox.h, FruitBox.cpp, FruitNode.h, LittleHand.h, LittleHand.cpp, Lime.h, Lime.cpp, Coconut.h, Coconut.cpp, Mixer.h, Mixer.cpp		
Notes: None		
Forbidden functions: 'using namespace' keyword		

Doesn't it make you feel kinda weird to be ingesting so many vitamins? I don't know about you, but it makes me wish I could change the world, create something new... But unlike the genetic modifications from **BioShock**, I promise to make them abide by **Google's** motto: *"Don't be evil"*.

I want these fruit to make the world a better, healthier place, where people can be truly happy. I want to modify `Fruits`, as you mya have guessed, to increase their vitaminic potential. I want more vitamins, always more! We'll have so many vitamins that our chakras will open, and just like when your mom tucks you into bed, the world will slowly be covered by a sweet and peaceful layer of whipped cream, topped with strawberries.

We'll find a way to modify our `Fruits` directly and conquer the world!

Once again, you'll use your `LittleHands` to perform this delicate operation. Add the following static member function to them:

```
void injectVitamin(Fruit &f, int quantity);
```

This function will inject (replace) `quantity` vitamins into `f`. Try using the following structure to modify matter itself, like **Neo**:

```
struct InTheMatrixFruit
{
    virtual ~InTheMatrixFruit();
    int vitamins;
};
```

You are now almost equal to the great gods of C++! You have the power to modify matter! The world and future are yours!