1 Tracker

This exercise covers: *special member functions*, *static data members*, *class templates*, *variadic templates*, *function templates* and *forwarding* (optional).

One big aspect of programming in C++ is managing memory and reducing the amount of copies created. Since a lot of copying is performed by the compiler through copy constructors and copy assignment operators, it can be hard to keep track of how many instances are present in the program at any given time. One way to make debugging these issues easier is to create a counter which is increased every time an object is created and decreased every time an object is destroyed. This will help us check if we have memory leaks and/or unnecessary instances.

In this exercise you are to implement a class template `Tracker` that takes a type template parameter, inherits from the supplied type and overloads the copy constructor, copy assignment operator and destructor.

`Tracker` should contain a `static` data member `counter` which is incremented when an object is created (or copied), and decremented when an object is destroyed.

It is important that `Tracker<T>` inherits from `T` because the entire public interface of `T` should be accessible through `Tracker<T>`, so that it can be used exactly as the type it tracks. However, this introduces a constraint on `T`, namely that `T` must be a *class type*, since those are the only types where inheritance is possible.

A general constructor should be implemented for `Tracker` which takes arbitrary arguments and passes them to the base class constructor (the constructor of `T`).
Extra: Try to use perfect forwarding for this constructor.

You should also implement a function template `count` that takes one type template parameter and returns how many objects of that type is currently tracked.

In the file `tracker.h` a test program is given, it demonstrate how the `Tracker` class template can be used.

2 Variadic List Operations

This exercise covers: *variadic templates*, *function templates*, *variadic recursion* and similar techniques.

Variadic parameter packs are very general and can in some sense act as lists of values or types during compile-time.

In this exercise you are to implement functionality to check if a variadic parameter pack contains a specific type and at which index in the pack. Basically search functionality for variadic packs.

To do this, first create an empty class template `Pack` that takes arbitrarily many types (a variadic template). This class template will "store" our type list so that we can pass

it to functions. Note that this class should not have any members whatsoever. This is important because it allows the class to be completely optimized away by the compiler.

You should then create two function templates: `contains` and `index_of`. These functions should both take one template parameter `T` and a function parameter `Pack` (remember that `Pack` must take an arbitrary number of arbitrary types).

`contains<T>(p)`, where `p` is some instantatiation of `Pack`, should return `true` if the variadic pack in `p` contains the type `T` and `false` otherwise.

`index_of<T>(p)` should return the index of the first occurance of `T` in `p`. If `T` does not occur, return this function should return `-1`.

Example: Assume that we have the following object `Pack<int, float, char> p`.

- `contains<int>(p)` should return `true`
- `contains<bool>(p)` should return `false`
- `index_of<int>(p)` should return `0`
- `index_of<char>(p)` should return `2`
- `index_of<bool>(p)` should return `-1`

Note that it should also be possible to write: `contains<int>(Pack<int, float, char>{})`.

Hint: Variadic recursion is the easiest way to solve this exercise.

3 Print the arguments

This exercise covers: *variadic templates*, *variadic recursion* and *fold expressions* (optional).

One of the original reasons for introducing functions with an arbitrary number of arguments was to make printing values easier. In C there is a function called `printf` that prints several arguments to `stdout`. C handles these parameters during runtime rather than compile-time, so the types of the arguments are lost. Therefore, the user must specify the types in a so called *format string* that is passed as the first value.

Example: `printf("%d %s %c", 5, "hello", 'a')` will print: `5 hello a`

In C++ we can do the same thing, but better! No need for a format string since we can generate an appropriate function during compile-time, while the types of the parameters are still known.

The goal of this exercise is to create a function `print` which takes an arbitrary amount of parameters with arbitrary types and prints them to the terminal seperated by a space.

`print` should work for all types that are *printable*, i.e. for types which can be printed to `cout` with `operator<<`.

Example: `print(5, "hello", 'a')` should print: `5 hello a`

There are some tests given in `print_test.cc`.
Hint: You can use variadic recursion or fold expressions to solve this exercise.