Introduction

In this take-home exam, you will write a series of routines that will perform basic actions on iterables. An *iterable* represents a sequence of values. All iterable instances are read-only, you cannot update an iterable. However, it is possible to create new iterables from existing ones.

An iterable i can be of finite or infinite size. Each value in i is called an *element* of i. Since i is a sequence, its elements have an order. Although the iterable interface does not allow indexing, for ease of description in this text, we will use $i[x]$ to refer to the xth element of i where x is a 0-based index. In other words, the iterable i is a sequence of the form:

$$i[0]\ ,\ i[1]\ ,\ i[2]\ ,\ \ldots$$

In this text, we will also use $len(i)$ to refer to the number of elements in i. It is possible that $len(i) = \infty$.

We read the elements of an iterable by creating an *iterator*, via the iterable's $iter()$ function. The elements can then be iterated through by repeatedly calling the $next()$ function on the iterator. To be more precise, a newly created iterator points to the first element of the iterable. Each $next()$ call returns the pointed element and advances the iterator to the next element. When all elements of the iterable are exhausted, the iterator throws a *StopIteration* exception upon a $next()$ call (and it should continue throwing the same exception for further $next()$ calls).

It is possible to create multiple iterators on a single iterable. Each iterator iterates through the iterable's elements independently. In other words, one can iterate through an iterable's elements more than once and simultaneously.

The following interface represents an iterable:

```
template<typename T>
class IIterable
{
    public:
        // Returns a new iterator for this iterable.
        virtual IIterator<T> * iter() const = 0;

        // Destroys the iterable.
        virtual ~IIterable() {};
};
```

The following interface represents an iterator:

```
template<typename T>
class IIterator
{
    public:
        // Returns the element pointed by this iterator and advances
        // to the next element in the iterable. If the iteration has
        // already reached the end of the iterable, this call throws
        // a StopIteration exception.
        virtual T next() = 0;

        // Destroys the iterator.
        virtual ~IIterator() {};
};
```

Finally, the following defines the StopIteration exception:

```
class StopIteration : public std::exception {};
```

Task

Iterables are created and manipulated via a set of global function templates which you are going to implement. Here is the list of the functions you need to implement:

- `template<typename T> IIterable<T> * repeat(T v);`
  Creates an iterable representing the infinite sequence

$$v, v, v, \ldots$$

- `template<typename T> IIterable<T> * count(T start, T delta);`
  Creates an iterable representing the infinite sequence

$$start\ ,\ start + delta\ ,\ start + delta + delta\ ,\ \ldots$$

- `template<typename T> IIterable<T> * skip(unsigned k, IIterable<T> *i);`
  Skips the first k elements of i. This would correspond to the sequence

$$i[k]\ ,\ i[k+1]\ ,\ i[k+2]\ ,\ \ldots$$

  If $k \geq len(i)$, then the resulting iterable is empty. The result has infinite length if $len(i) = \infty$.

- `template<typename T> IIterable<T> * take(unsigned k, IIterable<T> *i);`
  Creates an iterable representing the first k elements of i. That is,

$$i[0]\ ,\ \ldots,\ ,\ i[k-1]$$

  The resulting iterable is always finite. If $len(i) \leq k$ elements, then the result contains all elements of i.

- `template<typename T> IIterable<T> * cycle(IIterable<T> *i);`
  Creates an iterable that cycles through i infinitely many times. That is, it represents the sequence

$$i[0]\ ,\ \ldots\ ,\ i[len(i)-1]\ ,\ i[0]\ ,\ \ldots\ ,\ i[len(i)-1]\ ,\ i[0]\ ,\ \ldots$$

  If i is empty then so is the resulting iterable. If $len(i) = \infty$, then the result is equivalent to i.

- `template<typename T> IIterable<T> * concat(IIterable<T> *i, IIterable<T> *j);`
  Concatenates two iterables i and j. That is,

$$i[0]\ ,\ \ldots\ ,\ i[len(i)-1]\ ,\ j[0]\ ,\ j[1]\ ,\ \ldots$$

  If $len(i) = \infty$, then the result is equivalent to i.

- `template<typename T> IIterable<std::pair<T, T>> * pair(IIterable<T> *i);`
  Groups the elements of an iterable i in pairs. That is,

$$(i[0], i[1])\ ,\ (i[2], i[3])\ ,\ (i[4], i[5])\ ,\ \ldots$$

  If $len(i)$ is odd, then i's last element is ignored. Use `std::pairs` from the `<utility>` header to represent pairs.

- `template<typename T> IIterable<T> * min(IIterable<T> *i, IIterable<T> *j);`
  Creates an iterable that applies an element-wise min operation on two iterables i and j. That is,

$$min(i[0], j[0])\ ,\ min(i[1], j[1])\ ,\ \ldots$$

  If i and j have different lengths, the iteration finishes when either i or j is exhausted.

- `template<typename T> IIterable<T> * select(IIterable<T> *i, IIterable<unsigned> *s);`
  Selects a sequence of elements from i based on the indices provided in s. That is,

$$i[s[0]]\ ,\ i[s[1]]\ ,\ i[s[2]]\ ,\ \ldots$$

  If any index in s is invalid (i.e., $\geq len(i)$), then it is ignored and skipped. If $len(s) = \infty$, then the length of the resulting iterable may be infinite.

Example Code

To ease your thinking process, we will provide you an implementation of the following function:

- `template<typename T> IIterable<T> * accumulate(IIterable<T> *i);`
  Creates an iterable representing

$$i[0]\ ,\ i[0]+i[1]\ ,\ i[0]+i[1]+i[2]\ ,\ \ldots$$

Template Parameter

The template parameter T will be guaranteed to support:

- Default and copy construction.
- Assignment operator $=$.
- Consistent $+$, $+=$, $<$, $>$, $<=$ and $>=$ operators.

Ownership Semantics

The usage of the iterable and iterator objects is subject to a particular *ownership semantics*. At any time, an iterable/iterator object is owned by an *owner* code. This owner code is the sole code that can work on the object, but is also responsible for destroying the object when the object is no longer needed.

The following are the rules that describe the ownership:

- When an iterable is created via one of your functions, the caller becomes the owner of the returned iterable.
- When an iterable is passed to one of your functions as a parameter, your code becomes the new owner of the passed iterable. Since the caller code is losing ownership, it will guarantee that all iterators on the iterable have been destroyed at the time of calling. That also means that your code is now responsible for destroying the iterable. Of course, you cannot destroy the passed iterable right away; it will have to be used by the iterable that you return. (You cannot copy the elements of that iterable to your new iterable. Guess why?) The iterable that you return should keep track of the passed iterable and destroy it when it itself is destroyed.
- For functions that take two iterable parameters, the caller guarantees that they are different instances.
- When an iterator is created via $iter()$, the owner of the iterator is the caller. The ownership cannot be passed around, therefore the caller of $iter()$ is responsible ensuring the proper destruction of the iterator.

The main reason to define an ownership semantics is to ensure that all unneeded memory is properly and safely deallocated. In other words, it prevents memory leaks. Your code is responsible for following the rules above and deleting all owned objects that are no longer needed. Our grader will follow the rules and appropriately delete all objects that is owned by our code. If we detect a memory leak in your implementation, you may lose a portion of the points.

Lazy Evaluation

Your iterables and iterators should not do unnecessary data operations. We expect your iterators to behave *lazily*, retrieving just enough data from their dependencies to construct the result of the $next()$ call. (See how our example code achieves this.) Your code should avoid iterating through an iterable to store its elements in a large container (such as an array). Doing otherwise may cause you to lose points.

Submission

Submission is through ODTÜClass. The iterable interfaces and the $accumulate()$ function will be given to you in the *Iterable.tpp* file. We expect a single *the1.tpp* from you as part of the submission. This file should contain the implementation of your function templates and any additional classes necessary for your implementations. Ideally, your file should start with

```
#pragma once
#include "Iterable.tpp"
```

Your code will be compiled on g++ with the options: `-std=c++2a -w`. However, we suggest using the following options when compiling your code yourself: `-std=c++2a -Wall -Wextra -Wpedantic`.

Grading

You will submit your file through a VPL item in ODTÜClass. The item will have an "Evaluate" feature which you can use to auto-grade your submission. To get full marks, you need to implement all functions and your implementations should adhere to the description above. You will be able to get partial points if you implement a subset of the functions. Also note that your solution should not be "unreasonably" inefficient, in which case your code may hit the time/memory limits in place.

The grade from the auto-grader is not final. We may do additional black-box and white-box evaluations and adjust your grade. Solutions that do not reasonably attempt to solve the given task as described may lose points.

No late submissions are allowed. Please, start early. The logic you need to write is simple once you get the gist, however, the amount of code may still be large.

Example

The following code:

```
#include "Iterable.tpp"
#include "the1.tpp"
#include <iostream>

template<typename T>
std::ostream & operator<<(std::ostream & out, const std::pair<T, T> &p)
{
    return out << "(" << p.first << "," << p.second << ")";
}

template<typename T> void printAndDestroy(IIterable<T> *i)
{
    IIterator<T> *iterator = i->iter();

    try
    {
        while(true)
        {
            std::cout << iterator->next() << " ";
        }
    }
    catch (StopIteration &)
    {
        std::cout << std::endl;
    }

    delete iterator;
    delete i;
}

int main()
{
    printAndDestroy(take(3, repeat(0.f)));
    printAndDestroy(take(5, skip(3, count(1.f, 2.f))));
    printAndDestroy(take(5, accumulate(count(1.f, 2.f))));
    printAndDestroy(take(8, cycle(take(3, count(1.f, 1.f)))));
    printAndDestroy(concat(take(1, repeat(4.f)),
                           take(1, repeat(2.f))));
    printAndDestroy(pair(take(5, count(1.f, 1.f))));
    printAndDestroy(take(5, min(count(4.f, 1.f), count(1.f, 3.f))));
    printAndDestroy(select(count(10.f, 10.f),
                           concat(take(3, count(1u, 2u)),
                                  take(3, count(2u, 2u)))));
    return 0;
}
```

is expected to produce the output:

```
0 0 0
7 9 11 13 15
1 4 9 16 25
1 2 3 1 2 3 1 2
4 2
(1,2) (3,4)
1 4 6 7 8
20 40 60 30 50 70
```

Good luck!