# Números Naturales

Clase muy simple para iterar en el rango $n = \{0, 1, ..., n-1\}$. Otras clases la utilizan.

```
#include <algorithm>
#include <cassert>
#include <iostream>
#include <vector>

template <class IntType>
class basic_natural_number
{
public:
    using difference_type = long long;
    using size_type = long long;
    using value_type = IntType;
    class iterator;
    using const_iterator = iterator;

public:
    explicit basic_natural_number(IntType n) : m_n(n) { assert(n >= 0); }

    size_type size() const { return m_n; }

    class iterator
    {
    public:
        using iterator_category = std::random_access_iterator_tag;
        using value_type = IntType;
        using difference_type = long long;
        using pointer = IntType const*;
        using reference = const IntType&;

        explicit iterator(IntType t = 0) : m_ID(t) {}

        inline iterator& operator++()
        {
            ++m_ID;
            return *this;
        }
        inline iterator& operator--()
        {
            --m_ID;
            return *this;
        }
```

```
        inline const IntType& operator*() const { return m_ID; }

        inline iterator& operator+=(difference_type n)
        {
            m_ID += n;
            return *this;
        }

        inline iterator& operator-=(difference_type n)
        {
            return operator+=(-n);
        }

        inline bool operator==(const iterator& it) { return *it == m_ID; }

        inline bool operator!=(const iterator& it) { return *it != m_ID; }

        inline difference_type operator-(const iterator& it)
        {
            return *it - m_ID;
        }

    private:
        IntType m_ID{0};

        friend class basic_natural_number;
    }; // end class iterator

    iterator begin() const { return iterator(0); }

    iterator end() const { return iterator(m_n); }

    IntType operator[](size_type m) const { return m; }

    template <class Pred>
    IntType partition_point(Pred p)
    {
        return *std::partition_point(begin(), end(), p);
    }

private:
    IntType m_n;
}; // end class basic_natural_number

template <class IntType>
```

```
inline typename basic_natural_number<IntType>::iterator
operator+(typename basic_natural_number<IntType>::iterator it,
          typename basic_natural_number<IntType>::difference_type n)
{
    it += n;
    return it;
}

template <class IntType>
inline typename basic_natural_number<IntType>::iterator
operator-(typename basic_natural_number<IntType>::iterator it,
          typename basic_natural_number<IntType>::difference_type n)
{
    it -= n;
    return it;
}

using natural_number = basic_natural_number<int>;
using big_natural_number = basic_natural_number<long long>;

template <class Container, class T = typename Container::size_type>
basic_natural_number<T> indices(const Container& C)
{
    return basic_natural_number<T>(C.size());
}

int main()
{
    using std::cout;
    using std::endl;

    for (int i : natural_number(5))
        cout << i << ' ';

    cout << endl;

    std::vector<int> W = {2, 4, 6, 8};
    for (auto i : indices(W))
        cout << i << ": " << W[i] << endl;

    return 0;
}
```

**Output**:

```
0 1 2 3 4
```

```
0: 2
1: 4
2: 6
3: 8
```