```cpp
#include "prb/Interp.hpp"
#include "cfl/Error.hpp"

using namespace cfl;
using namespace prb;

// class Linear_Interp

class Linear_Interp_Function : public IFunction
{
public:
    Linear_Interp_Function(const std::vector<double> &rArg,
                           const std::vector<double> &rVal)
        : m_uArg(rArg), m_uVal(rVal)
    {
        POSTCONDITION(m_uArg.size() == m_uVal.size());
        POSTCONDITION(m_uArg.size() > 1);
        POSTCONDITION(std::equal(rArg.begin() + 1, rArg.end(), rArg.begin(),
                                 std::greater<double>()));
    }

    double operator()(double dX) const
    {
        PRECONDITION(belongs(dX));
        if (belongs(dX) == false)
        {
            throw(NError::range("linear interpolation"));
        }
        if (dX == m_uArg.front())
        {
            return m_uVal.front();
        }
        std::vector<double>::const_iterator itArg =
            std::lower_bound(m_uArg.begin(), m_uArg.end(), dX);
        std::vector<double>::const_iterator itVal =
            m_uVal.begin() + (itArg - m_uArg.begin());
        double dX2 = *itArg;
        double dY2 = *itVal;
        itArg--;
        itVal--;
        double dX1 = *itArg;
        double dY1 = *itVal;

        ASSERT(dX2 > dX1);

        return dY1 + (dY2 - dY1) * (dX - dX1) / (dX2 - dX1);
    }

    bool belongs(double dX) const
    {
        return (dX >= m_uArg.front()) && (dX <= m_uArg.back());
    }

private:
    std::vector<double> m_uArg, m_uVal;
};

class Linear_Interp : public IInterp
{
public:
    Function interpolate(const std::vector<double> &rArg,
                         const std::vector<double> &rVal) const
    {
        return Function(new Linear_Interp_Function(rArg, rVal));
    }
};

// functions from NInterp

cfl::Interp prb::NInterp::linear()
{
    return Interp(new Linear_Interp());
}
```

```cpp
#include "prb/Interp.hpp"
#include "cfl/Error.hpp"
#include <gsl/gsl_spline.h>

using namespace cfl;
using namespace prb;

// class Linear_Interp

class Linear_Interp_Function : public IFunction
{
public:
    Linear_Interp_Function(const std::vector<double> &rArg,
                           const std::vector<double> &rVal)
    {
        PRECONDITION(rArg.size() == rVal.size());
        PRECONDITION(rArg.size() > 1);
        PRECONDITION(std::equal(rArg.begin() + 1, rArg.end(), rArg.begin(),
                                std::greater<double>()));

        m_pSpline = gsl_spline_alloc(gsl_interp_linear, rArg.size());
        gsl_spline_init(m_pSpline, rArg.data(), rVal.data(), rArg.size());
        m_pAcc = gsl_interp_accel_alloc();
        m_dLeft = rArg.front();
        m_dRight = rArg.back();

        POSTCONDITION(m_dLeft < m_dRight);
    }

    ~Linear_Interp_Function()
    {
        gsl_spline_free(m_pSpline);
        gsl_interp_accel_free(m_pAcc);
    }

    double operator()(double dX) const
    {
        bool bBelongs = belongs(dX);
        PRECONDITION(bBelongs);
        if (!bBelongs)
        {
            throw(NError::range("interpolation"));
        }
        return gsl_spline_eval(m_pSpline, dX, m_pAcc);
    }

    bool belongs(double dX) const
    {
        return (dX >= m_dLeft) && (dX <= m_dRight);
    }

private:
    gsl_spline *m_pSpline;
    gsl_interp_accel *m_pAcc;
    double m_dLeft, m_dRight;
};

class Linear_Interp : public IInterp
{
public:
    Function interpolate(const std::vector<double> &rArg,
                         const std::vector<double> &rVal) const
    {
        return Function(new Linear_Interp_Function(rArg, rVal));
    }
};

// functions from NInterp

cfl::Interp prb::NInterp::linear()
{
    return Interp(new Linear_Interp());
}
```

```cpp
#include "prb/Interp.hpp"
#include "cfl/Error.hpp"
#include <gsl/gsl_spline.h>

using namespace cfl;
using namespace prb;

// class Linear_Interp

class Linear_Interp_Function : public IFunction
{
public:
    Linear_Interp_Function(const std::vector<double> &rArg,
                           const std::vector<double> &rVal)
        : m_uSpline(gsl_spline_alloc(gsl_interp_linear, rArg.size()),
                    &gsl_spline_free),
          m_uAcc(gsl_interp_accel_alloc(), &gsl_interp_accel_free)
    {
        PRECONDITION(rArg.size() == rVal.size());
        PRECONDITION(rArg.size() > 1);
        PRECONDITION(std::equal(rArg.begin() + 1, rArg.end(), rArg.begin(),
                                std::greater<double>()));

        gsl_spline_init(m_uSpline.get(), rArg.data(), rVal.data(), rArg.size());
        m_dLeft = rArg.front();
        m_dRight = rArg.back();

        POSTCONDITION(m_dLeft < m_dRight);
    }

    double operator()(double dX) const
    {
        bool bBelongs = belongs(dX);
        PRECONDITION(bBelongs);
        if (!bBelongs)
        {
            throw(NError::range("interpolation"));
        }
        return gsl_spline_eval(m_uSpline.get(), dX, m_uAcc.get());
    }

    bool belongs(double dX) const
    {
        return (dX >= m_dLeft) && (dX <= m_dRight);
    }

private:
    std::shared_ptr<gsl_spline> m_uSpline;
    std::shared_ptr<gsl_interp_accel> m_uAcc;
    double m_dLeft, m_dRight;
};

class Linear_Interp : public IInterp
{
public:
    Function interpolate(const std::vector<double> &rArg,
                         const std::vector<double> &rVal) const
    {
        return Function(new Linear_Interp_Function(rArg, rVal));
    }
};

// functions from NInterp

cfl::Interp prb::NInterp::linear()
{
    return Interp(new Linear_Interp());
}
```

```
#include "prb/Interp.hpp"
#include "cfl/Error.hpp"
#include <gsl/gsl_spline.h>

using namespace cfl;
using namespace prb;

// class Linear_Interp

class Linear_Interp_Function : public IFunction
{
public:
    Linear_Interp_Function(const std::vector<double> &rArg,
                           const std::vector<double> &rVal)
        : m_uSpline(gsl_spline_alloc(gsl_interp_linear, rArg.size()),
                    &gsl_spline_free),
          m_uAcc(gsl_interp_accel_alloc(), &gsl_interp_accel_free)
    {
        PRECONDITION(rArg.size() == rVal.size());
        PRECONDITION(rArg.size() > 1);
        PRECONDITION(std::equal(rArg.begin()+1, rArg.end(), rArg.begin(), std::greater<double>())
);

        gsl_spline_init(m_uSpline.get(), rArg.data(), rVal.data(), rArg.size());
        m_dLeft = rArg.front();
        m_dRight = rArg.back();

        POSTCONDITION(m_dLeft < m_dRight);
    }

    double operator()(double dX) const
    {
        bool bBelongs = belongs(dX);
        PRECONDITION(bBelongs);
        if (!bBelongs)
        {
            throw(NError::range("interpolation"));
        }
        return gsl_spline_eval(m_uSpline.get(), dX, m_uAcc.get());
    }

    bool belongs(double dX) const
    {
        return (dX >= m_dLeft) && (dX <= m_dRight);
    }

private:
    std::unique_ptr<gsl_spline, decltype(&gsl_spline_free)> m_uSpline;
    std::unique_ptr<gsl_interp_accel, decltype(&gsl_interp_accel_free)> m_uAcc;
    double m_dLeft, m_dRight;
};

class Linear_Interp : public IInterp
{
public:
    Function
    interpolate(const std::vector<double> &rArg,
                const std::vector<double> &rVal) const
    {
        return Function(new Linear_Interp_Function(rArg, rVal));
    }
};

// functions from NInterp

cfl::Interp prb::NInterp::linear()
{
    return Interp(new Linear_Interp());
}
```