EECE7205
Final Project Source Code
Vaibhav Kejriwal
(002201423)

MAIN.CPP

```
#include<iostream>
#include<vector>
#include<limits.h>
#include"Matrix.h"
#include<ctime>

using namespace std;
using namespace Numeric_lib;

struct task {
	int number; // number of task
	bool isCloudTask;  // judge whether the task is a cloud task
	double priority;
	int finishTimeLocal;  // finish time of the task in a core
	int finishTimeSending; // finish time of the task in sending
	int finishTimeCloud;   // finish time of the task in cloud
	int finishTimeReceiving;   // finish time of the task in receiving
	int readyTimeLocal; // earlist  time that the task can start in local core
	int readyTimeSending; // earlist  time that the task can start in sending
	int readyTimeCloud;  // earlist  time that the task can start in cloud
	int readyTimeReceiving; // earlist  time that the task can start in receiving
	int startTime; // task's actual start time
	int channel; // illustrate which channel the task operate (local core = 0,1,2,
cloud=3)
	bool isExitTask;  //whether it is an exit task
	bool isEntryTask;  // whether it is an entry task
	int readyCount1;
	int readyCount2;
};

// Phase one in step one: primary assignment
void primary(vector<task>&ini, Matrix<int, 2>&ta,int t)
{
	int min;
	unsigned int i;
	unsigned int j;
	for (i = 0; i < ta.dim1(); i++)
```

```
	{
		ini[i].number = i + 1;
		min = ta(i, 0);
		for (j = 0; j < ta.dim2(); j++)
			if (ta(i, j) < min)
				min = ta(i, j);
		if (min > t)
			ini[i].isCloudTask = 1;
		else
			ini[i].isCloudTask = 0;
	}
}

// Phase two in step one: task prioritizing
void prioritize(vector<task>&ini, Matrix<int, 2>&ta, Matrix<int, 2>&G,int t)
{
	unsigned int i;
	unsigned int j,m;
	int k ;
	double w;
	double max;
	m = ini.size() - 1;
	for (i = 0; i < ini.size(); i++)
	{
		k = 0;
		for (j = 0; j < G.dim2(); j++)
			if (G(m - i, j) == 1)
				k = k + 1;
		if (k == 0)
			ini[m - i].isExitTask = 1;
		k = 0;
		for (j = 0; j < G.dim2(); j++)
			if (G(j, m - i) == 1)
				k = k + 1;
		if(k==0)
			ini[m - i].isEntryTask = 1;
		max = 0;
		w = 0;
		if (!(ini[m - i].isCloudTask))
		{
			for (j = 0; j < ta.dim2(); j++)
				w = w + ta(m - i, j);
			w = w / 3;
		}
		else
			w = t;
```

```
			for (j = 0; j < G.dim2(); j++)
				if ((G(m - i, j) == 1) && (max < ini[j].priority))
					max = ini[j].priority;
			ini[m - i].priority = w + max;
		}
}

int find_biggest_pri(vector<task>&ini)
{
		unsigned int i;
		int max=0;
		for (i = 0; i < ini.size(); i++)
			if (ini[i].priority > ini[max].priority)
				max = i;
		return max;
}

// find the max in two numbers
int max2(int &m, int &n)
{
		if (m >= n)
			return m;
		else
			return n;
}

// if local schedule, return RTL
int d_rtl(task &vi, vector<task>&S, Matrix<int, 2>&G)
{
		unsigned int i;
		unsigned int j;
		int max=0;
		if (S.size()!=0)
		{
			for (i = 0; i < G.dim2(); i++)
				if (G(i, vi.number - 1) == 1)
					for (j = 0; j < S.size(); j++)
						if ((S[j].number == i + 1)&&(max <
max2(S[j].finishTimeLocal, S[j].finishTimeReceiving)))
							max = max2(S[j].finishTimeLocal,
S[j].finishTimeReceiving);
		}
		return max;
}

// if cloud schedule, return RTWS
```

```
int d_rtws(task &vi, vector<task>&S, Matrix<int, 2>&G)
{
	unsigned int i;
	unsigned int j;
	int max=0;
	if (S.size()!=0)
	{
		for (i = 0; i < G.dim2(); i++)
			if (G(i, vi.number - 1) == 1)
				for (j = 0; j < S.size(); j++)
					if ((S[j].number == i + 1)&&(max <
max2(S[j].finishTimeLocal, S[j].finishTimeSending)))
						max = max2(S[j].finishTimeLocal,
S[j].finishTimeSending);
	}
	return max;
}

// if cloud schedule, return RTC
int d_rtc(task &vi, vector<task>&S, Matrix<int, 2>&G)
{
	unsigned int i;
	unsigned int j;
	int max=vi.finishTimeSending;
	if (S.size()!=0)
	{
		for (i = 0; i < G.dim2(); i++)
			if (G(i, vi.number - 1) == 1)
				for (j = 0; j < S.size(); j++)
					if ((S[j].number == i + 1)&&(max <
max2(vi.finishTimeSending, S[j].finishTimeCloud)))
						max = max2(vi.finishTimeSending,
S[j].finishTimeCloud);
	}
	return max;
}

// if cloud schedule, return RTWR
int d_rtwr(task &vi)
{
	return vi.finishTimeCloud;
}

// if local schedule, return the smallest finish time
int locals(task &vi, vector<task>&S, Matrix<int, 2>&G, Matrix<int, 2>&ta)
{
```

```
	vi.readyTimeLocal = d_rtl(vi, S, G);
	unsigned int i;
	unsigned int j;
	int mint=INT_MAX;
	int ft;
	int max = 0; // find a local core's biggest finish time
	if (S.size()==0)
	{
		for (i = 0; i < ta.dim2(); i++)
		{
			ft = ta(vi.number - 1, i);
			if (mint > ft)
			{
				mint = ft;
				vi.channel = i;
			}
		}
		return mint;
	}
	for (i = 0; i < ta.dim2(); i++)
	{
		ft = vi.readyTimeLocal + ta(vi.number - 1, i);
		max = 0;
		for (j = 0; j < S.size(); j++)
			if ((S[j].channel == i) && (max < S[j].finishTimeLocal))
				max = S[j].finishTimeLocal;
		if(max>vi.readyTimeLocal)
			ft=max+ ta(vi.number - 1, i);
		if (mint > ft)
		{
			mint = ft;
			vi.channel = i;
		}
	}
	return mint;
}

// if cloud schedule, return the finish time
int clouds(task &vi, vector<task>&S, Matrix<int, 2>&G, int ts, int tc, int tr)
{
	vi.readyTimeSending = d_rtws(vi, S, G);
	unsigned int i;
	int maxs = 0;
	int t;
	int maxc = 0;
	int maxr = 0;
```

```
	int ft;
	t = ts + tc + tr;
	if (S.size()==0)
	{
		vi.finishTimeSending = ts;
		vi.readyTimeCloud = ts;
		vi.finishTimeCloud = ts + tc;
		vi.readyTimeReceiving = ts+tc;
		return t;
	}
	for(i=0;i<S.size();i++)
		if (S[i].channel == 3)
			if (maxs < S[i].finishTimeSending)
				maxs = S[i].finishTimeSending;
	if (maxs > vi.readyTimeSending)
		vi.finishTimeSending = maxs + ts;
	else
		vi.finishTimeSending = vi.readyTimeSending + ts;
	vi.readyTimeCloud = d_rtc(vi, S, G);
	for (i = 0; i < S.size(); i++)
		if (S[i].channel == 3)
			if (maxc < S[i].finishTimeCloud)
				maxc = S[i].finishTimeCloud;
	if (maxc > vi.readyTimeCloud)
		vi.finishTimeCloud = maxc + tc;
	else
		vi.finishTimeCloud = vi.readyTimeCloud + tc;
	vi.readyTimeReceiving = d_rtwr(vi);
	for (i = 0; i < S.size(); i++)
		if (S[i].channel == 3)
			if (maxr < S[i].finishTimeReceiving)
				maxr = S[i].finishTimeReceiving;
	if (maxr > vi.readyTimeReceiving)
		ft = maxr + tr;
	else
		ft = vi.readyTimeReceiving + tr;
	return ft;
}

void initials(vector<task>&S, vector<task>&ini, Matrix<int, 2>&ta, Matrix<int, 2>&G, int
ts, int tc, int tr)
{
	unsigned int i;
	int t;
	int maxp; // find the max priority in each iteration of ini
	int mint;  // find the minimum finish time of local
```

```
        int anot;  // perpare for another time (cloud)
        t = ts + tc + tr;
        for (i = 0; i < G.dim1(); i++)
        {
                maxp = find_biggest_pri(ini);
                if (!ini[maxp].isCloudTask)
                {
                        mint = locals(ini[maxp], S, G, ta);
                        anot = clouds(ini[maxp], S, G, ts, tc, tr);
                        if (anot < mint)
                        {
                                ini[maxp].readyTimeLocal = 0;
                                ini[maxp].finishTimeLocal = 0;
                                ini[maxp].channel = 3;
                                ini[maxp].finishTimeReceiving = anot;
                                ini[maxp].startTime = anot - t;
                        }
                        else
                        {
                                ini[maxp].finishTimeCloud = 0;
                                ini[maxp].finishTimeSending = 0;
                                ini[maxp].readyTimeSending = 0;
                                ini[maxp].readyTimeCloud = 0;
                                ini[maxp].readyTimeReceiving = 0;
                                ini[maxp].finishTimeReceiving = 0;
                                ini[maxp].finishTimeLocal = mint;
                                ini[maxp].startTime = mint - ta(ini[maxp].number - 1,
ini[maxp].channel);
                        }
                }
                else
                {
                        ini[maxp].finishTimeLocal = 0;
                        ini[maxp].readyTimeLocal = 0;
                        ini[maxp].channel = 3;
                        ini[maxp].finishTimeReceiving= clouds(ini[maxp], S, G, ts, tc, tr);
                        ini[maxp].startTime = ini[maxp].finishTimeReceiving - t;
                }
                S.push_back(ini[maxp]);
                ini.erase(ini.begin() + maxp);
        }
}

// return a task's finish time
int find_ft(task&vi)
{
```

```
        int max;
        max = max2(vi.finishTimeLocal, vi.finishTimeReceiving);
        return max;
 }

// print the sequence S
void prints(vector<task>&S)
{
        unsigned int i;
        int k,m;
        for (i = 0; i < S.size(); i++)
        {
                k = 1 + S[i].channel;
                m = find_ft(S[i]);
                cout << "Task" << S[i].number << ": ";
                switch (S[i].channel)
                {
                case 0:
                        cout << "local core" << k << ", ";
                        break;
                case 1:
                        cout << "local core" << k << ", ";
                        break;
                case 2:
                        cout << "local core" << k << ", ";
                        break;
                case 3:
                        cout << "cloud" << ", ";
                        break;
                default:
                        break;
                }
                cout << "start time is: " << S[i].startTime << ",finish time is: "<<m<<endl;
        }
}

// return the completion time of sequence S
int find_tcom(vector<task>&S)
{
        unsigned int i;
        int max=0;
        for (i = 0; i < S.size(); i++)
                if ((S[i].isExitTask) && (max < find_ft(S[i])))
                        max = find_ft(S[i]);
        return max;
}
```

```
// return the total energy of the sequence S
double find_en(vector<task>&S, int p1, int p2, int p3, double ps)
{
	unsigned int i;
	double ene=0;
	for (i = 0; i < S.size(); i++)
	{
		switch (S[i].channel)
		{
		case 0:
			ene = ene + p1 * (find_ft(S[i]) - S[i].startTime);
			break;
		case 1:
			ene = ene + p2 * (find_ft(S[i]) - S[i].startTime);
			break;
		case 2:
			ene = ene + p3 * (find_ft(S[i]) - S[i].startTime);
			break;
		case 3:
			ene = ene + ps * (S[i].finishTimeSending - S[i].startTime);
			break;
		default:
			break;
		}
	}
	return ene;
}

//compute all the ready1 in a sequence
void get_ready1(vector<task>&S, Matrix<int, 2>&G)
{
	unsigned int i, j, k;
	int m;
	for (i = 0; i < S.size(); i++)
	{
		m = 0;
		for (j = 0; j < G.dim2(); j++)
			if (G(j, S[i].number-1) == 1)
				for (k = 0; k < S.size(); k++)
					if (S[k].number == j + 1)
						m = m + 1;
		S[i].readyCount1 = m;
	}
}
```

```
//compute all the ready2 in a sequence
void get_ready2(vector<task>&S)
{
	unsigned int i, j;
	int m;
	for (i = 0; i < S.size(); i++)
	{
		m = 0;
		for (j = 0; j < S.size(); j++)
			if ((S[i].channel == S[j].channel) && (S[j].startTime < S[i].startTime))
				m = m + 1;
		S[i].readyCount2 = m;
	}
}

// local schedule task vi whose local core is confirmed
int localse(task &vi, vector<task>&SN, Matrix<int, 2>&G, Matrix<int, 2>&ta)
{
	vi.readyTimeLocal = d_rtl(vi, SN, G);
	unsigned int i;
	int ft;
	int max=0;
	if (SN.size()==0)
		ft = vi.readyTimeLocal + ta(vi.number - 1, vi.channel);
	else
	{
		for (i = 0; i < SN.size(); i++)
			if ((SN[i].channel == vi.channel) && (max < SN[i].finishTimeLocal))
				max = SN[i].finishTimeLocal;
		if(max>vi.readyTimeLocal)
			ft=max+ ta(vi.number - 1, vi.channel);
		else
			ft=vi.readyTimeLocal+ ta(vi.number - 1, vi.channel);
	}
	return ft;
}

void kernel(vector<task>&S, vector<task>&SN, int ktar, task vtar, Matrix<int, 2>&G,
Matrix<int, 2>&ta,int ts, int tc, int tr)
{
	unsigned int i;
	int m;
	int t;
	t = ts + tc + tr;
	vector<task>re;
	re = S;
```

```
	for (i = 0; i < re.size(); i++)
		if (vtar.number == re[i].number)
		{
			re[i].channel = ktar;
			if (ktar == 3)
			{
				re[i].finishTimeLocal = 0;
				re[i].readyTimeLocal = 0;
			}
		}
	while (re.size()!=0)
	{
		get_ready1(re, G);
		get_ready2(re);
		m = 0;
		while ((re[m].readyCount1!=0)&&(re[m].readyCount2 != 0))
			m = m + 1;
		if (re[m].channel == 3)
		{
			re[m].finishTimeReceiving = clouds(re[m], SN, G, ts, tc, tr);
			re[m].startTime = re[m].finishTimeReceiving - t;
		}
		else
		{
			re[m].finishTimeLocal = localse(re[m], SN, G, ta);
			re[m].startTime = re[m].finishTimeLocal - ta(re[m].number - 1,
re[m].channel);
		}
		SN.push_back(re[m]);
		re.erase(re.begin() + m);
	}
}

void mcc(vector<task>&S, Matrix<int, 2>&G, Matrix<int, 2>&ta, int ts, int tc, int tr,int p1,
int p2, int p3, double ps, int tmax)
{
	unsigned int i, j;
	int tcom;
	int tcom2;
	int a;
	double en;
	double en1;
	double en2;
	double ratio1=0;
	double ratio2;
	vector<task>SN;
```

```
        tcom = find_tcom(S);
        en = find_en(S, p1, p2, p3, ps);
        for (i = 0; i < S.size(); i++)
        {
                a = S[i].channel;
                if (S[i].channel != 3)
                {
                        for (j = 0; j < 4; j++)
                        {
                                if (j != a)
                                {
                                        SN.erase(SN.begin(), SN.end());
                                        en1 = find_en(S, p1, p2, p3, ps);
                                        kernel(S, SN, j, S[i], G, ta, ts, tc, tr);
                                        tcom2 = find_tcom(SN);
                                        en2 = find_en(SN, p1, p2, p3, ps);
                                        if ((en2 < en1) && (tcom >= tcom2))
                                                S = SN;
                                        else if ((en2 < en1) && (tcom2 <= tmax))
                                        {
                                                ratio2 = (en - en2) / (tcom2 - tcom);
                                                if (ratio2 > ratio1)
                                                {
                                                        ratio1 = ratio2;
                                                        S = SN;
                                                }
                                        }
                                }
                        }
                }
        }
}

void outerloop(vector<task>&S, Matrix<int, 2>&G, Matrix<int, 2>&ta, int ts, int tc, int tr,
int p1, int p2, int p3, double ps, int tmax)
{
        double en;
        double en1=0;
        en = find_en(S, p1, p2, p3, ps);
        while (en1<en)
        {
                en= find_en(S, p1, p2, p3, ps);
                mcc(S, G, ta, ts, tc, tr, p1, p2, p3, ps, tmax);
                en1= find_en(S, p1, p2, p3, ps);
        }
}
```

```
int main()
{
        int NUM_TASKS_GRAPH1 = 10;  // the number of tasks
        int NUM_LOCAL_CORES = 3;   // the number of local cores
        int ts1 = 3,tc1 = 1,tr1 = 1;
        unsigned int i;
        int t1;
        int tmax1 = 27;
        int PRIORITY_LOCAL_CORE_1 = 1;
        int PRIORITY_LOCAL_CORE_2 = 2;
        int PRIORITY_LOCAL_CORE_3 = 4;
       double PRIORITY_CLOUD = 0.5;
       double rt;
       clock_t start, end;
        t1 = ts1 + tc1 + tr1;
        Matrix<int, 2>G1(NUM_TASKS_GRAPH1,NUM_TASKS_GRAPH1);
        Matrix<int, 2>ta1(NUM_TASKS_GRAPH1,NUM_LOCAL_CORES);
        vector<task>ini1(NUM_TASKS_GRAPH1);
        vector<task>S1;
        G1(0, 1) = 1;
        G1(0, 2) = 1;
        G1(0, 3) = 1;
        G1(0, 4) = 1;
        G1(0, 5) = 1;
        G1(1, 7) = 1;
        G1(1, 8) = 1;
        G1(2, 6) = 1;
        G1(3, 7) = 1;
        G1(3, 8) = 1;
        G1(4, 8) = 1;
        G1(5, 7) = 1;
        G1(6, 9) = 1;
        G1(7, 9) = 1;
        G1(8, 9) = 1;




        ta1(0, 0) = 9;
        ta1(0, 1) = 7;
        ta1(0, 2) = 5;
        ta1(1, 0) = 8;
        ta1(1, 1) = 6;
        ta1(1, 2) = 5;
        ta1(2, 0) = 6;
```

```
ta1(2, 1) = 5;
ta1(2, 2) = 4;
ta1(3, 0) = 7;
ta1(3, 1) = 5;
ta1(3, 2) = 3;
ta1(4, 0) = 5;
ta1(4, 1) = 4;
ta1(4, 2) = 2;
ta1(5, 0) = 7;
ta1(5, 1) = 6;
ta1(5, 2) = 4;
ta1(6, 0) = 8;
ta1(6, 1) = 5;
ta1(6, 2) = 3;
ta1(7, 0) = 6;
ta1(7, 1) = 4;
ta1(7, 2) = 2;
ta1(8, 0) = 5;
ta1(8, 1) = 3;
ta1(8, 2) = 2;
ta1(9, 0) = 7;
ta1(9, 1) = 4;
ta1(9, 2) = 2;

primary(ini1, ta1, t1);
prioritize(ini1, ta1, G1,t1);
start = clock();
initials(S1, ini1, ta1, G1, ts1, tc1, tr1);
end = clock();
cout << "Initial schedule: " << endl;
prints(S1);
rt = (double)(end - start) / (double)(CLOCKS_PER_SEC)*(double)(1000.000000);
cout << " Now the total energy is: " << find_en(S1, PRIORITY_LOCAL_CORE_1,
PRIORITY_LOCAL_CORE_2, PRIORITY_LOCAL_CORE_3, PRIORITY_CLOUD)<<endl;
cout << " Now the completion time is: " << find_tcom(S1) << endl;
cout << "Running time of initial schedule of Graph is "<<rt<<" ms"<< endl;
start = clock();
outerloop(S1, G1, ta1, ts1, tc1, tr1, PRIORITY_LOCAL_CORE_1,
PRIORITY_LOCAL_CORE_2, PRIORITY_LOCAL_CORE_3, PRIORITY_CLOUD, tmax1);
end = clock();
cout << "After Task Migration: " << endl;
prints(S1);
rt = (double)(end - start) / (double)(CLOCKS_PER_SEC)*(double)(1000.000000);
cout << " Now the total energy is: " << find_en(S1, PRIORITY_LOCAL_CORE_1,
PRIORITY_LOCAL_CORE_2, PRIORITY_LOCAL_CORE_3, PRIORITY_CLOUD) << endl;
cout << " Now the completion time is: " << find_tcom(S1) << endl;
```

```
        cout << "Running time of task migration of Graph is "<<rt<<" ms"<< endl;
        cout<<endl;
}
```

MATRIX.H

```
/*
   warning: this small multidimensional matrix library uses a few features
   not taught in ENGR112 and not explained in elementary textbooks

   (c) Bjarne Stroustrup, Texas A&M University.

   Use as you like as long as you acknowledge the source.
*/

#ifndef MATRIX_LIB
#define MATRIX_LIB

#include<string>
#include<algorithm>
//#include<iostream>

namespace Numeric_lib {

//-----------------------------------------------------------------------------

struct Matrix_error {
   std::string name;
   Matrix_error(const char* q) :name(q) { }
   Matrix_error(std::string n) :name(n) { }
};

//-----------------------------------------------------------------------------

inline void error(const char* p)
{
   throw Matrix_error(p);
}

//-----------------------------------------------------------------------------

typedef long Index;    // I still dislike unsigned
```

```
//-----------------------------------------------------------------------------

// The general Matrix template is simply a prop for its specializations:
template<class T = double, int D = 1> class Matrix {
    // multidimensional matrix class
    // ( ) does multidimensional subscripting
    // [ ] does C style "slicing": gives an N-1 dimensional matrix from an N dimensional
one
    // row() is equivalent to [ ]
    // column() is not (yet) implemented because it requires strides.
    // = has copy semantics
    // ( ) and [ ] are range checked
    // slice() to give sub-ranges
private:
    Matrix();    // this should never be compiled
//        template<class A> Matrix(A);
};

//-----------------------------------------------------------------------------

template<class T = double, int D = 1> class Row ;    // forward declaration

//-----------------------------------------------------------------------------

// function objects for various apply() operations:

template<class T> struct Assign {
    void operator()(T& a, const T& c) { a = c; }
};

template<class T> struct Add_assign {
    void operator()(T& a, const T& c) { a += c; }
};
template<class T> struct Mul_assign {
    void operator()(T& a, const T& c) { a *= c; }
};
template<class T> struct Minus_assign {
    void operator()(T& a, const T& c) { a -= c; }
};
template<class T> struct Div_assign {
    void operator()(T& a, const T& c) { a /= c; }
};
template<class T> struct Mod_assign {
    void operator()(T& a, const T& c) { a %= c; }
};
```

```
template<class T> struct Or_assign {
    void operator()(T& a, const T& c) { a |= c; }
};
template<class T> struct Xor_assign {
    void operator()(T& a, const T& c) { a ^= c; }
};
template<class T> struct And_assign {
    void operator()(T& a, const T& c) { a &= c; }
};

template<class T> struct Not_assign {
    void operator()(T& a) { a = !a; }
};

template<class T> struct Not {
    T operator()(T& a) { return !a; }
};

template<class T> struct Unary_minus {
    T operator()(T& a) { return -a; }
};

template<class T> struct Complement {
    T operator()(T& a) { return ~a; }
};

//-----------------------------------------------------------------------------

// Matrix_base represents the common part of the Matrix classes:
template<class T> class Matrix_base {
    // matrixs store their memory (elements) in Matrix_base and have copy semantics
    // Matrix_base does element-wise operations
protected:
    T* elem;    // vector? no: we couldn't easily provide a vector for a slice
    const Index sz;
    mutable bool owns;
    mutable bool xfer;
public:
    Matrix_base(Index n) :elem(new T[n]()), sz(n), owns(true), xfer(false)
        // matrix of n elements (default initialized)
    {
        // std::cerr << "new[" << n << "]->" << elem << "\n";
    }

    Matrix_base(Index n, T* p) :elem(p), sz(n), owns(false), xfer(false)
        // descriptor for matrix of n elements owned by someone else
```

```
{
}

~Matrix_base()
{
   if (owns) {
      // std::cerr << "delete[" << sz << "] " << elem << "\n";
      delete[]elem;
   }
}

// if necessay, we can get to the raw matrix:
     T* data()       { return elem; }
const T* data() const { return elem; }
Index    size() const { return sz; }

void copy_elements(const Matrix_base& a)
{
   if (sz!=a.sz) error("copy_elements()");
   for (Index i=0; i<sz; ++i) elem[i] = a.elem[i];
}

void base_assign(const Matrix_base& a) { copy_elements(a); }

void base_copy(const Matrix_base& a)
{
   if (a.xfer) {          // a is just about to be deleted
                        // so we can transfer ownership rather than copy
      // std::cerr << "xfer @" << a.elem << " [" << a.sz << "]\n";
      elem = a.elem;
      a.xfer = false;    // note: modifies source
      a.owns = false;
   }
   else {
      elem = new T[a.sz];
      // std::cerr << "base copy @" << a.elem << " [" << a.sz << "]\n";
      copy_elements(a);
   }
   owns = true;
   xfer = false;
}

// to get the elements of a local matrix out of a function without copying:
void base_xfer(Matrix_base& x)
{
   if (owns==false) error("cannot xfer() non-owner");
```

```
		owns = false;	// now the elements are safe from deletion by original owner
		x.xfer = true;	// target asserts temporary ownership
		x.owns = true;
	}

	template<class F> void base_apply(F f) { for (Index i = 0; i<size(); ++i) f(elem[i]); }
	template<class F> void base_apply(F f, const T& c) { for (Index i = 0; i<size(); ++i)
f(elem[i],c); }
private:
	void operator=(const Matrix_base&);	// no ordinary copy of bases
	Matrix_base(const Matrix_base&);
};

//-----------------------------------------------------------------------------

template<class T> class Matrix<T,1> : public Matrix_base<T> {
	const Index d1;

protected:
	// for use by Row:
	Matrix(Index n1, T* p) : Matrix_base<T>(n1,p), d1(n1)
	{
		// std::cerr << "construct 1D Matrix from data\n";
	}

public:

	Matrix(Index n1) : Matrix_base<T>(n1), d1(n1) { }

	Matrix(Row<T,1>& a) : Matrix_base<T>(a.dim1(),a.p), d1(a.dim1())
	{
		// std::cerr << "construct 1D Matrix from Row\n";
	}

	// copy constructor: let the base do the copy:
	Matrix(const Matrix& a) : Matrix_base<T>(a.size(),0), d1(a.d1)
	{
		// std::cerr << "copy ctor\n";
		this->base_copy(a);
	}

	template<int n>
	Matrix(const T (&a)[n]) : Matrix_base<T>(n), d1(n)
		// deduce "n" (and "T"), Matrix_base allocates T[n]
	{
		// std::cerr << "matrix ctor\n";
```

```
        for (Index i = 0; i<n; ++i) this->elem[i]=a[i];
    }

    Matrix(const T* p, Index n) : Matrix_base<T>(n), d1(n)
        // Matrix_base allocates T[n]
    {
        // std::cerr << "matrix ctor\n";
        for (Index i = 0; i<n; ++i) this->elem[i]=p[i];
    }

    template<class F> Matrix(const Matrix& a, F f) : Matrix_base<T>(a.size()), d1(a.d1)
        // construct a new Matrix with element's that are functions of a's elements:
        // does not modify a unless f has been specifically programmed to modify its
argument
        // T f(const T&) would be a typical type for f
    {
        for (Index i = 0; i<this->sz; ++i) this->elem[i] = f(a.elem[i]);
    }

    template<class F, class Arg> Matrix(const Matrix& a, F f, const Arg& t1) :
Matrix_base<T>(a.size()), d1(a.d1)
        // construct a new Matrix with element's that are functions of a's elements:
        // does not modify a unless f has been specifically programmed to modify its
argument
        // T f(const T&, const Arg&) would be a typical type for f
    {
        for (Index i = 0; i<this->sz; ++i) this->elem[i] = f(a.elem[i],t1);
    }

    Matrix& operator=(const Matrix& a)
        // copy assignment: let the base do the copy
    {
        // std::cerr << "copy assignment (" << this->size() << ',' << a.size()<< ")\n";
        if (d1!=a.d1) error("length error in 1D=");
        this->base_assign(a);
        return *this;
    }

    ~Matrix() { }

    Index dim1() const { return d1; }    // number of elements in a row

    Matrix xfer()    // make an Matrix to move elements out of a scope
    {
        Matrix x(dim1(),this->data()); // make a descriptor
        this->base_xfer(x);                  // transfer (temporary) ownership to x
```

```
    return x;
}

void range_check(Index n1) const
{
    // std::cerr << "range check: (" << d1 << "): " << n1 << "\n";
    if (n1<0 || d1<=n1) error("1D range error: dimension 1");
}

// subscripting:
      T& operator()(Index n1)       { range_check(n1); return this->elem[n1]; }
const T& operator()(Index n1) const { range_check(n1); return this->elem[n1]; }

// slicing (the same as subscripting for 1D matrixs):
      T& operator[](Index n)       { return row(n); }
const T& operator[](Index n) const { return row(n); }

      T& row(Index n)       { range_check(n); return this->elem[n]; }
const T& row(Index n) const { range_check(n); return this->elem[n]; }

Row<T,1> slice(Index n)
    // the last elements from a[n] onwards
{
    if (n<0) n=0;
    else if(d1<n) n=d1;// one beyond the end
    return Row<T,1>(d1-n,this->elem+n);
}

const Row<T,1> slice(Index n) const
    // the last elements from a[n] onwards
{
    if (n<0) n=0;
    else if(d1<n) n=d1;// one beyond the end
    return Row<T,1>(d1-n,this->elem+n);
}

Row<T,1> slice(Index n, Index m)
    // m elements starting with a[n]
{
    if (n<0) n=0;
    else if(d1<n) n=d1;    // one beyond the end
    if (m<0) m = 0;
    else if (d1<n+m) m=d1-n;
    return Row<T,1>(m,this->elem+n);
}
```

```
const Row<T,1> slice(Index n, Index m) const
    // m elements starting with a[n]
{
    if (n<0) n=0;
    else if(d1<n) n=d1;    // one beyond the end
    if (m<0) m = 0;
    else if (d1<n+m) m=d1-n;
    return Row<T,1>(m,this->elem+n);
}

// element-wise operations:
template<class F> Matrix& apply(F f) { this->base_apply(f); return *this; }
template<class F> Matrix& apply(F f,const T& c) { this->base_apply(f,c); return
*this; }

Matrix& operator=(const T& c)  { this->base_apply(Assign<T>(),c);       return *this; }

Matrix& operator*=(const T& c) { this->base_apply(Mul_assign<T>(),c);   return *this; }
Matrix& operator/=(const T& c) { this->base_apply(Div_assign<T>(),c);   return *this; }
Matrix& operator%=(const T& c) { this->base_apply(Mod_assign<T>(),c);   return
*this; }
Matrix& operator+=(const T& c) { this->base_apply(Add_assign<T>(),c);   return *this;
}
Matrix& operator-=(const T& c) { this->base_apply(Minus_assign<T>(),c); return *this;
}

Matrix& operator&=(const T& c) { this->base_apply(And_assign<T>(),c);   return *this;
}
Matrix& operator|=(const T& c) { this->base_apply(Or_assign<T>(),c);    return *this; }
Matrix& operator^=(const T& c) { this->base_apply(Xor_assign<T>(),c);   return
*this; }

Matrix operator!() { return xfer(Matrix(*this,Not<T>())); }
Matrix operator-() { return xfer(Matrix(*this,Unary_minus<T>())); }
Matrix operator~() { return xfer(Matrix(*this,Complement<T>()));  }

template<class F> Matrix apply_new(F f) { return xfer(Matrix(*this,f)); }

void swap_rows(Index i, Index j)
    // swap_rows() uses a row's worth of memory for better run-time performance
    // if you want pairwise swap, just write it yourself
{
    if (i == j) return;
/*
    Matrix<T,1> temp = (*this)[i];
    (*this)[i] = (*this)[j];
```

```
        (*this)[j] = temp;
    */
        Index max = (*this)[i].size();
        for (Index ii=0; ii<max; ++ii) std::swap((*this)(i,ii),(*this)(j,ii));
    }
};

//-----------------------------------------------------------------------------

template<class T> class Matrix<T,2> : public Matrix_base<T> {
    const Index d1;
    const Index d2;

protected:
    // for use by Row:
    Matrix(Index n1, Index n2, T* p) : Matrix_base<T>(n1*n2,p), d1(n1), d2(n2)
    {
      //  std::cerr << "construct 3D Matrix from data\n";
    }

public:

    Matrix(Index n1, Index n2) : Matrix_base<T>(n1*n2), d1(n1), d2(n2) { }

    Matrix(Row<T,2>& a) : Matrix_base<T>(a.dim1()*a.dim2(),a.p), d1(a.dim1()),
d2(a.dim2())
    {
      // std::cerr << "construct 2D Matrix from Row\n";
    }

    // copy constructor: let the base do the copy:
    Matrix(const Matrix& a) : Matrix_base<T>(a.size(),0), d1(a.d1), d2(a.d2)
    {
        // std::cerr << "copy ctor\n";
        this->base_copy(a);
    }

    template<int n1, int n2>
    Matrix(const T (&a)[n1][n2]) : Matrix_base<T>(n1*n2), d1(n1), d2(n2)
        // deduce "n1", "n2" (and "T"), Matrix_base allocates T[n1*n2]
    {
        // std::cerr << "matrix ctor (" << n1 << "," << n2 << ")\n";
        for (Index i = 0; i<n1; ++i)
            for (Index j = 0; j<n2; ++j) this->elem[i*n2+j]=a[i][j];
    }
```

```
    template<class F> Matrix(const Matrix& a, F f) : Matrix_base<T>(a.size()), d1(a.d1),
d2(a.d2)
        // construct a new Matrix with element's that are functions of a's elements:
        // does not modify a unless f has been specifically programmed to modify its
argument
        // T f(const T&) would be a typical type for f
    {
        for (Index i = 0; i<this->sz; ++i) this->elem[i] = f(a.elem[i]);
    }

    template<class F, class Arg> Matrix(const Matrix& a, F f, const Arg& t1) :
Matrix_base<T>(a.size()), d1(a.d1), d2(a.d2)
        // construct a new Matrix with element's that are functions of a's elements:
        // does not modify a unless f has been specifically programmed to modify its
argument
        // T f(const T&, const Arg&) would be a typical type for f
    {
        for (Index i = 0; i<this->sz; ++i) this->elem[i] = f(a.elem[i],t1);
    }

    Matrix& operator=(const Matrix& a)
        // copy assignment: let the base do the copy
    {
        // std::cerr << "copy assignment (" << this->size() << ',' << a.size()<< ")\n";
        if (d1!=a.d1 || d2!=a.d2) error("length error in 2D =");
        this->base_assign(a);
        return *this;
    }

    ~Matrix() { }

    Index dim1() const { return d1; }    // number of elements in a row
    Index dim2() const { return d2; }    // number of elements in a column

    Matrix xfer()    // make an Matrix to move elements out of a scope
    {
        Matrix x(dim1(),dim2(),this->data()); // make a descriptor
        this->base_xfer(x);            // transfer (temporary) ownership to x
        return x;
    }

    void range_check(Index n1, Index n2) const
    {
        // std::cerr << "range check: (" << d1 << "," << d2 << "): " << n1 << " " << n2 <<
"\n";
        if (n1<0 || d1<=n1) error("2D range error: dimension 1");
```

```
        if (n2<0 || d2<=n2) error("2D range error: dimension 2");
    }

    // subscripting:
          T& operator()(Index n1, Index n2)       { range_check(n1,n2); return this-
>elem[n1*d2+n2]; }
    const T& operator()(Index n1, Index n2) const { range_check(n1,n2); return this-
>elem[n1*d2+n2]; }

    // slicing (return a row):
          Row<T,1> operator[](Index n)       { return row(n); }
    const Row<T,1> operator[](Index n) const { return row(n); }

          Row<T,1> row(Index n)       { range_check(n,0); return Row<T,1>(d2,&this-
>elem[n*d2]); }
    const Row<T,1> row(Index n) const { range_check(n,0); return Row<T,1>(d2,&this-
>elem[n*d2]); }

    Row<T,2> slice(Index n)
        // rows [n:d1)
    {
        if (n<0) n=0;
        else if(d1<n) n=d1;    // one beyond the end
        return Row<T,2>(d1-n,d2,this->elem+n*d2);
    }

    const Row<T,2> slice(Index n) const
        // rows [n:d1)
    {
        if (n<0) n=0;
        else if(d1<n) n=d1;    // one beyond the end
        return Row<T,2>(d1-n,d2,this->elem+n*d2);
    }

    Row<T,2> slice(Index n, Index m)
        // the rows [n:m)
    {
        if (n<0) n=0;
        if(d1<m) m=d1;    // one beyond the end
        return Row<T,2>(m-n,d2,this->elem+n*d2);

    }

    const Row<T,2> slice(Index n, Index m) const
        // the rows [n:sz)
    {
```

```
        if (n<0) n=0;
        if(d1<m) m=d1;    // one beyond the end
        return Row<T,2>(m-n,d2,this->elem+n*d2);
    }

    // Column<T,1> column(Index n); // not (yet) implemented: requies strides and
operations on columns

    // element-wise operations:
    template<class F> Matrix& apply(F f)            { this->base_apply(f);   return *this; }
    template<class F> Matrix& apply(F f,const T& c) { this->base_apply(f,c); return
*this; }

    Matrix& operator=(const T& c)  { this->base_apply(Assign<T>(),c);       return *this; }

    Matrix& operator*=(const T& c) { this->base_apply(Mul_assign<T>(),c);   return *this; }
    Matrix& operator/=(const T& c) { this->base_apply(Div_assign<T>(),c);   return *this; }
    Matrix& operator%=(const T& c) { this->base_apply(Mod_assign<T>(),c);   return
*this; }
    Matrix& operator+=(const T& c) { this->base_apply(Add_assign<T>(),c);   return *this;
}
    Matrix& operator-=(const T& c) { this->base_apply(Minus_assign<T>(),c); return *this;
}

    Matrix& operator&=(const T& c) { this->base_apply(And_assign<T>(),c);   return *this;
}
    Matrix& operator|=(const T& c) { this->base_apply(Or_assign<T>(),c);    return *this; }
    Matrix& operator^=(const T& c) { this->base_apply(Xor_assign<T>(),c);   return
*this; }

    Matrix operator!() { return xfer(Matrix(*this,Not<T>())); }
    Matrix operator-() { return xfer(Matrix(*this,Unary_minus<T>())); }
    Matrix operator~() { return xfer(Matrix(*this,Complement<T>()));  }

    template<class F> Matrix apply_new(F f) { return xfer(Matrix(*this,f)); }

    void swap_rows(Index i, Index j)
        // swap_rows() uses a row's worth of memory for better run-time performance
        // if you want pairwise swap, just write it yourself
    {
        if (i == j) return;
    /*
        Matrix<T,1> temp = (*this)[i];
        (*this)[i] = (*this)[j];
        (*this)[j] = temp;
    */
```

```
        Index max = (*this)[i].size();
        for (Index ii=0; ii<max; ++ii) std::swap((*this)(i,ii),(*this)(j,ii));
    }
};

//-----------------------------------------------------------------------------

template<class T> class Matrix<T,3> : public Matrix_base<T> {
    const Index d1;
    const Index d2;
    const Index d3;

protected:
    // for use by Row:
    Matrix(Index n1, Index n2, Index n3, T* p) : Matrix_base<T>(n1*n2*n3,p), d1(n1),
d2(n2), d3(n3)
    {
        // std::cerr << "construct 3D Matrix from data\n";
    }

public:

    Matrix(Index n1, Index n2, Index n3) : Matrix_base<T>(n1*n2*n3), d1(n1), d2(n2),
d3(n3) { }

    Matrix(Row<T,3>& a) : Matrix_base<T>(a.dim1()*a.dim2()*a.dim3(),a.p), d1(a.dim1()),
d2(a.dim2()), d3(a.dim3())
    {
        // std::cerr << "construct 3D Matrix from Row\n";
    }

    // copy constructor: let the base do the copy:
    Matrix(const Matrix& a) : Matrix_base<T>(a.size(),0), d1(a.d1), d2(a.d2), d3(a.d3)
    {
        // std::cerr << "copy ctor\n";
        this->base_copy(a);
    }

    template<int n1, int n2, int n3>
    Matrix(const T (&a)[n1][n2][n3]) : Matrix_base<T>(n1*n2), d1(n1), d2(n2), d3(n3)
        // deduce "n1", "n2", "n3" (and "T"), Matrix_base allocates T[n1*n2*n3]
    {
        // std::cerr << "matrix ctor\n";
        for (Index i = 0; i<n1; ++i)
            for (Index j = 0; j<n2; ++j)
                for (Index k = 0; k<n3; ++k)
```

```
                this->elem[i*n2*n3+j*n3+k]=a[i][j][k];
    }

    template<class F> Matrix(const Matrix& a, F f) : Matrix_base<T>(a.size()), d1(a.d1), 
d2(a.d2), d3(a.d3)
        // construct a new Matrix with element's that are functions of a's elements:
        // does not modify a unless f has been specifically programmed to modify its 
argument
        // T f(const T&) would be a typical type for f
    {
        for (Index i = 0; i<this->sz; ++i) this->elem[i] = f(a.elem[i]);
    }

    template<class F, class Arg> Matrix(const Matrix& a, F f, const Arg& t1) : 
Matrix_base<T>(a.size()), d1(a.d1), d2(a.d2), d3(a.d3)
        // construct a new Matrix with element's that are functions of a's elements:
        // does not modify a unless f has been specifically programmed to modify its 
argument
        // T f(const T&, const Arg&) would be a typical type for f
    {
        for (Index i = 0; i<this->sz; ++i) this->elem[i] = f(a.elem[i],t1);
    }

    Matrix& operator=(const Matrix& a)
        // copy assignment: let the base do the copy
    {
        // std::cerr << "copy assignment (" << this->size() << ',' << a.size()<< ")\n";
        if (d1!=a.d1 || d2!=a.d2 || d3!=a.d3) error("length error in 2D =");
        this->base_assign(a);
        return *this;
    }

    ~Matrix() { }

    Index dim1() const { return d1; }    // number of elements in a row
    Index dim2() const { return d2; }    // number of elements in a column
    Index dim3() const { return d3; }    // number of elements in a depth

    Matrix xfer()    // make an Matrix to move elements out of a scope
    {
        Matrix x(dim1(),dim2(),dim3(),this->data()); // make a descriptor
        this->base_xfer(x);            // transfer (temporary) ownership to x
        return x;
    }

    void range_check(Index n1, Index n2, Index n3) const
```

```
    {
       // std::cerr << "range check: (" << d1 << "," << d2 << "): " << n1 << " " << n2 <<
"\n";
       if (n1<0 || d1<=n1) error("3D range error: dimension 1");
       if (n2<0 || d2<=n2) error("3D range error: dimension 2");
       if (n3<0 || d3<=n3) error("3D range error: dimension 3");
    }

    // subscripting:
          T& operator()(Index n1, Index n2, Index n3)       { range_check(n1,n2,n3); return
this->elem[d2*d3*n1+d3*n2+n3]; };
    const T& operator()(Index n1, Index n2, Index n3) const { range_check(n1,n2,n3);
return this->elem[d2*d3*n1+d3*n2+n3]; };

    // slicing (return a row):
          Row<T,2> operator[](Index n)       { return row(n); }
    const Row<T,2> operator[](Index n) const { return row(n); }

          Row<T,2> row(Index n)       { range_check(n,0,0); return Row<T,2>(d2,d3,&this-
>elem[n*d2*d3]); }
    const Row<T,2> row(Index n) const { range_check(n,0,0); return
Row<T,2>(d2,d3,&this->elem[n*d2*d3]); }

    Row<T,3> slice(Index n)
       // rows [n:d1)
    {
       if (n<0) n=0;
       else if(d1<n) n=d1;    // one beyond the end
       return Row<T,3>(d1-n,d2,d3,this->elem+n*d2*d3);
    }

    const Row<T,3> slice(Index n) const
       // rows [n:d1)
    {
       if (n<0) n=0;
       else if(d1<n) n=d1;    // one beyond the end
       return Row<T,3>(d1-n,d2,d3,this->elem+n*d2*d3);
    }

    Row<T,3> slice(Index n, Index m)
       // the rows [n:m)
    {
       if (n<0) n=0;
       if(d1<m) m=d1;    // one beyond the end
       return Row<T,3>(m-n,d2,d3,this->elem+n*d2*d3);
```

```
    }

    const Row<T,3> slice(Index n, Index m) const
        // the rows [n:sz)
    {
        if (n<0) n=0;
        if(d1<m) m=d1;    // one beyond the end
        return Row<T,3>(m-n,d2,d3,this->elem+n*d2*d3);
    }

    // Column<T,2> column(Index n); // not (yet) implemented: requies strides and
operations on columns

    // element-wise operations:
    template<class F> Matrix& apply(F f)           { this->base_apply(f);   return *this; }
    template<class F> Matrix& apply(F f,const T& c) { this->base_apply(f,c); return
*this; }

    Matrix& operator=(const T& c)  { this->base_apply(Assign<T>(),c);       return *this; }

    Matrix& operator*=(const T& c) { this->base_apply(Mul_assign<T>(),c);   return *this; }
    Matrix& operator/=(const T& c) { this->base_apply(Div_assign<T>(),c);   return *this; }
    Matrix& operator%=(const T& c) { this->base_apply(Mod_assign<T>(),c);   return
*this; }
    Matrix& operator+=(const T& c) { this->base_apply(Add_assign<T>(),c);   return *this;
}
    Matrix& operator-=(const T& c) { this->base_apply(Minus_assign<T>(),c); return *this;
}

    Matrix& operator&=(const T& c) { this->base_apply(And_assign<T>(),c);   return *this;
}
    Matrix& operator|=(const T& c) { this->base_apply(Or_assign<T>(),c);    return *this; }
    Matrix& operator^=(const T& c) { this->base_apply(Xor_assign<T>(),c);   return
*this; }

    Matrix operator!() { return xfer(Matrix(*this,Not<T>())); }
    Matrix operator-() { return xfer(Matrix(*this,Unary_minus<T>())); }
    Matrix operator~() { return xfer(Matrix(*this,Complement<T>()));  }

    template<class F> Matrix apply_new(F f) { return xfer(Matrix(*this,f)); }

    void swap_rows(Index i, Index j)
        // swap_rows() uses a row's worth of memory for better run-time performance
        // if you want pairwise swap, just write it yourself
    {
        if (i == j) return;
```

```
        Matrix<T,2> temp = (*this)[i];
        (*this)[i] = (*this)[j];
        (*this)[j] = temp;
    }
};

//-----------------------------------------------------------------------------

template<class T> Matrix<T> scale_and_add(const Matrix<T>& a, T c, const
Matrix<T>& b)
    //  Fortran "saxpy()" ("fma" for "fused multiply-add").
    // will the copy constructor be called twice and defeat the xfer optimization?
{
    if (a.size() != b.size()) error("sizes wrong for scale_and_add()");
    Matrix<T> res(a.size());
    for (Index i = 0; i<a.size(); ++i) res[i] += a[i]*c+b[i];
    return res.xfer();
}

//-----------------------------------------------------------------------------

template<class T> T dot_product(const Matrix<T>&a , const Matrix<T>& b)
{
    if (a.size() != b.size()) error("sizes wrong for dot product");
    T sum = 0;
    for (Index i = 0; i<a.size(); ++i) sum += a[i]*b[i];
    return sum;
}

//-----------------------------------------------------------------------------

template<class T, int N> Matrix<T,N> xfer(Matrix<T,N>& a)
{
    return a.xfer();
}

//-----------------------------------------------------------------------------

template<class F, class A>            A apply(F f, A x)        { A res(x,f);   return xfer(res); }
template<class F, class Arg, class A> A apply(F f, A x, Arg a) { A res(x,f,a); return
xfer(res); }

//-----------------------------------------------------------------------------

// The default values for T and D have been declared before.
```

```
template<class T, int D> class Row {
   // general version exists only to allow specializations
private:
      Row();
};

//-----------------------------------------------------------------------

template<class T> class Row<T,1> : public Matrix<T,1> {
public:
   Row(Index n, T* p) : Matrix<T,1>(n,p)
   {
   }

   Matrix<T,1>& operator=(const T& c) { this->base_apply(Assign<T>(),c); return *this; }

   Matrix<T,1>& operator=(const Matrix<T,1>& a)
   {
      return *static_cast<Matrix<T,1>*>(this)=a;
   }
};

//-----------------------------------------------------------------------

template<class T> class Row<T,2> : public Matrix<T,2> {
public:
   Row(Index n1, Index n2, T* p) : Matrix<T,2>(n1,n2,p)
   {
   }

   Matrix<T,2>& operator=(const T& c) { this->base_apply(Assign<T>(),c); return *this; }

   Matrix<T,2>& operator=(const Matrix<T,2>& a)
   {
      return *static_cast<Matrix<T,2>*>(this)=a;
   }
};

//-----------------------------------------------------------------------

template<class T> class Row<T,3> : public Matrix<T,3> {
public:
   Row(Index n1, Index n2, Index n3, T* p) : Matrix<T,3>(n1,n2,n3,p)
   {
   }
```

```
    Matrix<T,3>& operator=(const T& c) { this->base_apply(Assign<T>(),c); return *this; }

    Matrix<T,3>& operator=(const Matrix<T,3>& a)
    {
        return *static_cast<Matrix<T,3>*>(this)=a;
    }
};

//-----------------------------------------------------------------------------

template<class T, int N> Matrix<T,N-1> scale_and_add(const Matrix<T,N>& a, const
Matrix<T,N-1> c, const Matrix<T,N-1>& b)
{
    Matrix<T> res(a.size());
    if (a.size() != b.size()) error("sizes wrong for scale_and_add");
    for (Index i = 0; i<a.size(); ++i) res[i] += a[i]*c+b[i];
    return res.xfer();
}

//-----------------------------------------------------------------------------

template<class T, int D> Matrix<T,D> operator*(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r*=c; }
template<class T, int D> Matrix<T,D> operator/(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r/=c; }
template<class T, int D> Matrix<T,D> operator%(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r%=c; }
template<class T, int D> Matrix<T,D> operator+(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r+=c; }
template<class T, int D> Matrix<T,D> operator-(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r-=c; }

template<class T, int D> Matrix<T,D> operator&(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r&=c; }
template<class T, int D> Matrix<T,D> operator|(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r|=c; }
template<class T, int D> Matrix<T,D> operator^(const Matrix<T,D>& m, const T& c)
{ Matrix<T,D> r(m); return r^=c; }

//-----------------------------------------------------------------------------

}
#endif
```