Mitarbeiterseminar(e)

“Jump–start C++”

“Trilinos used in DRT”

“DRT discretization management”

Michael W. Gee
Lehrstuhl für Numerische Mechanik
Prof. Dr.–Ing. Wolfgang A. Wall
TU München

Why c++ and Trilinos and a new discretization management?

- Our applications become much more complex and we need to reuse and integrate our groups research achievements to a much larger extend then before
- We have to become ’organized’ meaning we need clearly defined modules and objects that we can easily combine to create new methods.

  (Xfem/Mortar/Chimera turbulaent FSI with thermo coupling in biomechanical tissue contact with noise scattering in the throat of whales or so....)
- Our current worst bottleneck is the inflexibility and inefficiency or the current discretization *(we will have a new worst but hopefully smaller bottleneck after this...)*
- Another bad bottleneck is the explicit dependency on GID.
- c++ is *one* appropiate language to pursue and achieve these goals (the other would be fortran90/95), and C is *not*. Python, Java, Matlab etc. are not suitable for one or the other reason, mainly performance.
- We outsource all parallel linear (and nonlinear?) algebra to Trilinos:
  - Trilinos is c++ at the user/developer level and fortran77/c at the kernel
  - Trilinos is LGPL
  - Trilinos is close to our style of parallel thinking (dom. decomp., MPI)
  - Trilinos is *very* powerful and *very* mature in all basic subpackages
- Integrating the mature Trilinos basic subpackages (Epetra, AztecOO, NOX, ML, Ifpack, Amesos, Teuchos) lets us easily play with the less mature but maybe useful other packages as well (Rythmos, Loca, Meros, Anaszasi, Thyra, PyTrilinos ....)

Some of the C++ features unfamiliar to C

`cout/new/delete/string/{}/bool/namespaces/reference/files/ overloading`

Classes

`basic classes / public, private, protected`
`virtual classes / pure virtual classes / inheritance`
`casting`

Templates & STL

`templates / vector / map / multimap / iterators`

Reference counting pointer and parameter list (Trilinos)

`RefCountPtr / ParameterList`

Epetra parallel linear algebra objects (Trilinos)

`Epetra_Comm / Epetra_Map / Epetra_Vector / Epetra_CrsMatrix`

```
// declaration
namespace DRT;
{
    void Myfunction(...);
    void Myfunction2(...);
}

// definition
void DRT::Myfunction(...)
{
    // whatever this function does
    return;
}

// usage:

DRT::Myfunction(...);

// or:

using namespace DRT;
Myfunction(...);
```

Namespaces

```
using namespace std;

// cout is a stream to stdout
cout << “Hello world\n”;
// or
cout << “Hello world” << endl;

// or
cout << “a + b = 1 : “ << 1 << “ + “ << 5 << “ = “ << 1+5 << endl;

// often, more abstract objects ’implement the ostream <<’ operation
// e.g.
ostream& operator << (ostream& os, const DRT::Element& ele)
{
    // printing the element here
}

// usage then
DRT::Element myelement;
cout << myelement;

// there is also stuff like cin, cerr etc.....

// often it is more convenient to do c-style printing

printf(”In %d cases, c is more convenient to print\n”,5);
```

std::cout

```
// c style malloc is 'new' in c++

int* ptr = new int[50];

// c style free is 'delete' in c++

delete [] ptr;

// CAREFUL:

// Allocate and delete an array of things
// (see above)

// Allocate and delete single object
DRT::Node* nodeptr = new Node(...);
delete nodeptr;

// do NOT mix delete  and  delete [] and c and c++ style

// you can NOT do
DRT::Node* nodeptr = new Node(...)[50];
delete [] nodeptr;
// -> works only with objects that have an 'empty constructor' like int
```

new / delete

comments

```
// c++ has a separate boolean variable type

bool wantthis = true;
...
if (wantthis)
{
  ...
}
...
wantthis = false;

// works of course as every other standard variable:

bool* boolvec = new bool[30];

boolvec[17] = false;

delete [] boolvec;

// (under the hood, its actually an integer and can be used as such)
```

bool

```cpp
// in a c++ code you can define a variable at any place
void myfunction(....)
{
    double a = 5.0;
    double b = 7.0;

    {
        double c = a+b;
    }

    double d = c; // this is wrong, c does not exist here

    for (int i=0; i<10; ++i)
    {
        double sum = a+b+d;
        sum *= i;
    }
}

// note that a variable exists only in the {} where it was declared
// -> c exists only inside its {}
// -> i exist only inside the loop
// -> sum exists only inside the loop (making this loop somewhat useless)
// -> the synthax for i helps the compiler determine that nobody messes with i
```

comments

```
void myfunction(....)
{
    {
      // c-style reference
      int i = 5;
      int* iptr;
      iptr = &i;
      *iptr = 7;
    }

    {
      // c++ style reference
      int i = 5;
      int& iref = i;
      iref = 7;
    }

    int j = 13;
    // wrong! not referencing
    // anything, won't compile
    int& jref;
    jref = j;
}
```

Note:

Unlike fortran, c only has call–by–value. If you need a call–by–reference, a call–by–value on the adress of an object is made

c++ has a true call by reference (reference operator &)

One can not have a reference variable not referencing anything. This way, a reference (unlike a pointer) is *always* well defined and valid.

references 1

comments

```
void myfunction(....)
{
  // c-style call-by-reference
  int i = 5;
  AddTwo(&i);
}

AddTwo(int* i)
{
  *i += 2;
  return;
}


void myfunction_cpp(....)
{
  // c++-style call-by-reference
  int i = 5;
  AddTwo2_cpp(i);
}

AddTwo_cpp(int& i)
{
  i += 2;
  return;
}
```

Don’t have to mess with all this pointer/adress business, just use a call by reference or a call by value

Cannot have bad references as one can have bad pointers

A reference is always ’good’ otherwise the compiler notices.

references 2

comments

files

node.H

```
// file description comment
#ifndef NODE_H
#define NODE_H

// some declarations
// doxygen docu of nodefunction1
void nodefunction1(...);
// doxygen docu of nodefunction2
int nodefunction2(...);

#endif // end of header file
```

node.cpp

```
// file description comment
#include “node.H”
#include”headthatincludesnodeaswell.H”

// short comment on nodefunction1
void nodefunction1(...)
{
    ...
}
// short comment on nodefunction 2
int nodefunction2(...)
{
    ...
}
```

multiple and nested inclusions of node.H are ok due to NODE_H

c++ does not tolerate multiple declarations

comments

node.H

```
// file description comment
#ifndef NODE_H
#define NODE_H

// some declarations
void TwiceThis(const int& i);
void TwiceThis(const double& i);


#endif // end of header file
```

node.cpp

```
// file description comment
#include “node.H”
#include”headthatincludesnodeaswell.H”

// short comment on AddThis
void TwiceThis(const int& i)
{
    i = i*2;
}
// short comment on AddThis
void TwiceThis(const double& i)
{
    i = i*2.0;
}
```

– Overloading can help save brain power (Just remember there was a TwiceThis method that worked for all reasonable data types)

– Be very reasonable, one can do a lot of nonsense with overloading..., e.g.

```
void TwiceThis(const double& i)
{
    i = i*i;
}
```

comments

Some of the C++ features unfamiliar to C

`cout/new/delete/string/{}/bool/namespaces/reference/files/`
`overloading`

<– We are here now –>

Classes

`basic classes / public, private, protected`
`virtual classes / pure virtual classes / inheritance`
`casting`

Templates & STL

`templates / vector / map / multimap / iterators`

Reference counting pointer and parameter list

`RefCountPtr / ParameterList`

Epetra parallel linear algebra objects

`Epetra_Comm / Epetra_Map / Epetra_Vector /`
`Epetra_CrsMatrix`

classes: basics 1 – public/private/protected

```cpp
// file contains class Animal
#ifndef ANIMAL_H
#define ANIMAL_H

// declaration of class
class Animal
{
public:

    // stuff that can be accessed from
    // outside the class
    int pubi_;

protected:

    // stuff that can be accessed from
    // under certain circumstances
    // (later)
    int protj_;

private:

    // stuff that is invisible to
    // the outside world
    int privk_;

}; // end of class Animal
#endif // end of header file
```

A class is something like a struct:
it can hold data (int, double, etc)
it can also hold functions
it protects its data and functions by
classifying it as public/protected/private

```cpp
#include “animal.H”

...
{
    // create an instance of Animal
    Animal myanimal;
    myanimal.pubi_ = 5; // ok

    // might be ok if i’m either
    // a friend of Animal
    // or derived from Animal here (later)
    myanimal.protj_ = 7;

    myanimal.privk_ = 13; // won’t compile
}
```

Private data can only be accessed (read/write)
by the class’ own functions

classes: basics 2 – methods

```
// file contains class Animal
#ifndef ANIMAL_H
#define ANIMAL_H

// declaration of class
class Animal
{
public:

    int Returnk()
    {
        return privk_;
    }

    int pubi_;

protected:

    int protj_;

private:

    int privk_;

}; // end of class Animal
#endif // end of header file
```

A function inside a class is called a method.
All methods can access private/protected data.

```
#include "animal.H"
...
{
    // create an instance of Animal
    Animal myanimal;
    myanimal.pubi_ = 5; // ok

    // might be ok if i'm either
    // a friend of Animal
    // or derived from Animal here (later)
    myanimal.protj_ = 7;

    // ok, note that k is a copy of
    // privk_ !
    double k = myanimal.Returnk();
}
```

Private/Protected data can only be accessed (read/write) by the class' own methods

One can also have private methods, that can only be called by other methods of the class

comments

```
// file contains class Animal
#ifndef ANIMAL_H
#define ANIMAL_H

// declaration of class
class Animal
{
public:

    int ReturnI() { return i_; }

    int* AccessI() { return &i_; }

    int ReturnJ() { return j_; }

    int& ReturnKRef() { return k_; }

protected:

private:

    int i_;
    int j_;
    int k_;

}; // end of class Animal
#endif // end of header file
```

Never grant unlimited public access to your data.
Data should be at least protected

```
#include “animal.H”
...
{
    // create an instance of Animal
    Animal myanimal;

    int i = myanimal.ReturnI();
    int* iptr = myanimal.AccessI();
    *iptr = 7; // nice
    *(myanimal.AccessI()) = 7; // nice?

    int& kref = myanimal.ReturnKRef();
    kref = 17; // nicer
    myanimal.ReturnKRef() = 17; // nice?
}
```

classes: basics 3 – good practice

comments

classes: basics 4 – ctor, cctor and dtor

```
class Animal
{
public:

  // constructor
  Animal(int j);

  // another constructor (empty)
  Animal();

  // copy constructor
  Animal(const Animal& olda);

  // destructor
  ~Animal();

  // some method (definition missing)
  MyMethod(int fool);

protected:
private:

  int j_;
  int* myvec_; // vector length j_

}; // end of class Animal
```

```
Animal::Animal(int j)
{
  j_ = j;
  myvec_ = new int[j_];
  return;
}

Animal::Animal()
{
  j_ = -1;
  myvec_ = NULL;
  return;
}

Animal::Animal(const Animal& olda)
{
  j_ = olda.j_;
  myvec_ = new int[j_];
  for (int i=0; i<j_; ++i)
    myvec_[i] = olda.myvec_[i];
  return;
}

Animal::~Animal()
{
  if (myvec_) delete [] myvec_;
  return;
}
```

classes: basics 5 – good practice

```
class Animal
{
public:

   // constructor
   Animal(int j);

   // another constructor (empty)
   Animal();

   // destructor
   ~Animal();

   // some method (definition missing)
   MyMethod(int fool);

protected:
private:

   // do not want copy-ctor
   Animal(const Animal& old);

   // do not want = operator
   Animal operator = (const Animal& old);

}; // end of class Animal
```

If you do not define
copy constructor and = operator,
the compiler will automatically generate
default versions of them which
might show unexpected behavior

–> declare private to be sure
    not to have them

comments

```
class Animal
{
public:

  // constructor
  Animal(int j);

  // another constructor (empty)
  Animal();

  // destructor
  ~Animal();

  // some method (definition missing)
  MyMethod(int fool);

protected:
private:

  // do not want copy-ctor
  Animal(const Animal& old);

  // do not want = operator
  Animal operator = (const Animal& old);

}; // end of class Animal
```

- namespaces are ALL capital, e.g.
  `namespace DISCRETIZATION`
- classes are captial and small, e.g.
  `class LineElement;`
- Methods and functions are capital and small, e.g.
  `int i = MyMethod();`
- variables and instances are ALL small, e.g.
  ```
  int i;
  Solid3 myelement(...);
  ```
- variables/instances inside classes end with an underscore, e.g.:
  ```
  class Animal
  {
    int i_;
    Node& mynode_;
  }
  ```

classes: basics 6 – more good practice

Design a class that describes a Person

```
class Person
{

}; // end of class Person
```

classes: basics 7 – Exercise

Create a class for an Element

```
class Element
{
  Element(int id, int nnode, int* nodeids);
  Element(const Element& old);
  virtual ~Element();

  virtual int Id()
  { return id_; }

  virtual int NumNode()
  { return nnode_; }

  virtual int* NodeIds()
  { return nodeids_; }


protected:

  int id_;
  int nnode_;
  int* nodeids_;

}; // end of class Element
```

A virtual class is basically a class that declares at least its destructor 'virtual'

Now, we need about x different types of elements (shell, fluid3, solid3, ale2 ...)

We could:

- store ALL of their specific data in Element
- Create a separate Element class for each of them and make the discretization handle them all (a lot of if's and switches)
- use derived classes
- other ways not discussed here...

classes: virtual/derived classes 1

Create a class for an Element

```
class Element
{
  Element(int id, int nnode, int* nodeids);
  Element(const Element& old);
  virtual ~Element();

  virtual int Id()
  { return id_; }

  virtual int NumNode()
  { return nnode_; }

  virtual int* NodeIds()
  { return nodeids_; }

protected:

  int id_;
  int nnode_;
  int* nodeids_;
}; // end of class Element
```

”base class”

```
Solid3::~Solid3()
{
  material_ = -1; // useless...
  return;
}
```

```
class Solid3 : public Element
{
  Solid3(int id, int nnode,
         int* nodeids, int material);

  Solid3(const Solid3& old);

  virtual ~Solid3();

  int Material()
  { return material_; }

protected:

  int material_;

}; // end of class Solid3
```

”derived class”

```
Solid3::Solid3(int id, int nnode,
        int* nodeids, int material) :
Element(id,nnode,nodeids)
{
  material_ = material;
}

Solid3::Solid3(const Solid3& old) :
Element(old)
{
  material_ = old.material_;
}
```

classes: virtual/derived classes 2

Create a class for an Element

```
class Element
{
  Element(int id, int nnode, int* nodeids);
  Element(const Element& old);
  virtual ~Element();

  virtual int Id()
  { return id_; }

  virtual int NumNode()
  { return nnode_; }

  virtual int* NodeIds()
  { return nodeids_; }

protected:

  int id_;
  int nnode_;
  int* nodeids_;
}; // end of class Element
```

”base class”

```
class Solid3 : public Element
{
  Solid3(int id, int nnode,
         int* nodeids, int material);

  Solid3(const Solid3& old);

  virtual ~Solid3();

  int Material()
  { return material_; }

protected:

  int material_;

}; // end of class Solid3
```

”derived class”

```
{
  int nnode = 4;
  int* ids = new int[nnode]; // fill this...
  Solid3 myele(1,nnode,ids,7);
  delete [] ids;

  int  eleid   = myele.Id();
  int  nn      = myele.NumNode();
  int* nodeids = myele.NodeIds();
  int  mat     = myele.Material();
}
```

classes: virtual/derived classes 3

comments

Create a class for an Element

```
class Element
{
  Element(int id, int nnode, int* nodeids);
  Element(const Element& old);
  virtual ~Element();

  virtual int Id()
  { return id_; }

  virtual int NumNode()
  { return nnode_; }

  virtual int* NodeIds()
  { return nodeids_; }

protected:
  int id_;
  int nnode_;
  int* nodeids_;
}; // end of class Element
```

”base class”

```
class Solid3 : public Element
{
  Solid3(int id, int nnode,
         int* nodeids, int material);

  Solid3(const Solid3& old);

  virtual ~Solid3();

  int* NodeIds() { return NULL; }

  int Material()
  { return material_; }

protected:

  int material_;

}; // end of class Solid3
```

”derived class”

Just looking at the Element part of Solid3:

```
{
  ...
  Solid3 myele(1,nnode,ids,7);
  Element& baseele = myele; // works fine
  // always returns NULL as well!
  int* nodeids = baseele.NodeIds();
}
```

A derived class can reimplement a base class method:

```
{
  ...
  Solid3 myele(1,nnode,ids,7);
  // always returns NULL !
  int* nodeids = myele.NodeIds();
}
```

classes: virtual/derived classes 4

Create a class for an Element

```
class Element
{
  Element(int id, int nnode, int* nodeids);
  Element(const Element& old);
  virtual ~Element();

  virtual int Id()
  { return id_; }

  virtual int NumNode()
  { return nnode_; }

  virtual int* NodeIds()
  { return nodeids_; }

protected:
  int id_;
  int nnode_;
  int* nodeids_;
}; // end of class Element
```

```
class Solid3 : public Element
{
  Solid3(int id, int nnode,
         int* nodeids, int material);

  Solid3(const Solid3& old);

  virtual ~Solid3();

  int* NodeIds() { return NULL; }
```

”

class type	function type	search order
derived	normal	derived, then base
base	normal	base
derived	virtual	derived, then base
base	virtual	derived, then base

Just looking at the Element part of Solid3:

```
{
  ...
  Solid3 myele(1,nnode,ids,7);
  Element& baseele = myele; // works fine
  // always returns NULL as well!
  int* nodeids = baseele.NodeIds();
}
```

A derived class can reimplement a base class method:

```
{
  ...
  Solid3 myele(1,nnode,ids,7);
  // always returns NULL !
  int* nodeids = myele.NodeIds();
}
```

classes: virtual/derived classes 4

comments

Create a base class for an Animal
(e.g. no. of legs)

Create a class for an elephant

Create a class for a honeybee

classes: virtual/derived classes Exercise

Create a base class for an Animal
(e.g. no. of legs)

```
class Animal
{
  Animal();
  Animal(const Animal& old);
  virtual ~Animal();

  virtual int Nlegs() = 0;
  (...) // other stuff
private:
  // do not want these
  Animal(const Animal& old);
  Animal operator = (const Animal& old);
}; // end of class Animal
```

’Animal’ is pure virtual,
one can not have an instance of Animal:

```
 {
   Animal myanim; // this will not compile
 }
```

Animal is ment to derive from, you can
not use Animal itself

Create a class for an elephant

```
class Elephant : public Animal
{
  Elephant();
  Elephant(const Elephant& old);
  virtual ~Elephant();

  int Nlegs() { return 4; }

private:
  // do not want these
  Elephant(const Elephant& old);
  Elephant operator = (const Elephant& old);
}; // end of class Elephant
```

Create a class for a honeybee

```
class Hbee : public Animal
{
  Hbee();
  Hbee(const Hbee& old);
  virtual ~Hbee();

  int Nlegs() { return 6; }

private:
  // do not want these
  Hbee(const Hbee& old);
  Hbee operator = (const Hbee& old);
}; // end of class Hbee
```

classes: pure virtual classes 5

comments

```
class Element
{
  Element(int id, int nnode, int* nodeids);
  Element(const Element& old);
  virtual ~Element();

  virtual int Id()
  { return id_; }

  virtual int NumNode()
  { return nnode_; }

  virtual int* NodeIds()
  { return nodeids_; }

protected:

  int id_;
  int nnode_;
  int* nodeids_;
}; // end of class Element
```

”base class”

```
class Solid3 : public Element
{
  Solid3(int id, int nnode,
         int* nodeids, int material);

  Solid3(const Solid3& old);

  virtual ~Solid3();

  int Material()
  { return material_; }

protected:

  int material_;

}; // end of class Solid3
```

”derived class”

```
{
  const Solid3 myele(1,nnode,ids,7);
  Solid3& eleref = const_cast<Solid3&>(myele); // bad, explicit cast-away constness
  Element& baseele = dynamic_cast<Element&>(eleref); // good
  Element* bptr = dynamic_cast<Element*>(&eleref); // good
  Element* bptr = static_cast<Element*>(&eleref); // brutal
  Element* bptr = (Element*)(&eleref); // c-style brutal
}
```

classes: type casting 6

A template is not a real piece of code, but an instruction to generate a piece of code if required:

```
template<typename T> void SwapTheseTwo(T& a, T& b)
{
  T tmp = a;
  a = b;
  b = tmp;
  return;
};

// a 'specialization' of SwapTheseTwo for 'Element'
template<> void SwapTheseTwo(Element& a, Element& b)
{
  // here swap the contents of the two elements a and b
  return;
}


{
  int i = 5;
  int j = 7;
  SwapTheseTwo(i,j);
  double x = 6.34;
  double y = 3.141;
  SwapTheseTwo(x,y);
  Element a(5,nodeids,false);
  Element b(7,nodeids2,true);
  SwapTheseTwo(a,b);
}
```

’Real’ code for <typename int> and <typename double> versions of “SwapTheseTwo” is generated by the compiler at link time and compiled.

For <Element> the compiler uses the supplied specialization.

templates 1

comments

A template is not a real piece of code, but an instruction to generate a piece of code if required:

```
template<typename T> T* Allocate(int num)
{
  T* data = new T[num];
  return data;
};
```

In cases, where it is not obvious from the parameter list of a function what type of variable was meant, one has to explicitly give the type where the function is used:

```
{
  // the compiler can't see from the parameters,
  // what we want
  double* dptr = Allocate<double>(10);
  int* iptr = Allocate<int>(12);
}
```

– One can have template functions, methods and classes
– Bugs in templates can be very hard to find, they are hard to debug and often produce very kryptic compiler/linker messages leaving no clue what the problem might be.
– Usage of templates (especially those from the STL) is desired in $CODENAME
– Creation of templates is in general not encouraged as they can be very hard to understand for (future) coworkers
– Create templates iff you feel absolutely comfortable with them!

templates 2

comments

c++ comes along with a series of very powerful and useful templates we should excessively use. These are bundled in the so called STL

```
#include <vector>

{
  // c- style array
  double* x = malloc(5*sizeof(double));
  if (!x) printf(”Error: Allocation failed\n”);
  free(x);

  // c++ conventional array
  double* x = new double[5];
  if (!x) cout << “Error: Allocation failed\n”;
  delete [] x;

  // c++ stl style array
  vector<double> x(5); // constructor with a given size
  x[0] = 3.245;
  x[1] = 5.897;
  x[2] = x[0] + x[1];
  x[4] = x[2] - x[1];
  // not really necessary:
  x.clear();
  // x dies correctly when the destructor of vector is
  // called, no memory leaking!
}
```

the Standard Template LIB (STL) 1

An incomplete list of STL containers:

```
{
  vector<typename T>                              <– we'll only talk about these

  map<typename key, typename datatype>

  set<typename T>

  list<typename T>

  multimap<typename key, typename datatype>

  hashmap<typename key, typename datatype>

  stack<typename T>
}
```

the Standard Template LIB (STL) 2

vector

```
#include <vector>
{
  vector<int> ivec; // vector of length 0
  vector<double> dvec(10) // vector length 10;

  ivec.resize(50); // now length 50;
  int ilength = ivec.size(); // get length

  // iterate through ivec
  for (int i=0; i<(int)ivec.size(); ++i)
  {
    ivec[i] = i*i;
  }

  // get direct access to continuous data in dvec
  double* dptr = &devc[0];
  // iterate though devc in an uncommon style
  for (int i=0; i<(int)devc.size(); ++i)
  {
    dptr[i] = 3.141*i;
  }

}
// all vectors died correctly here
```

the Standard Template LIB (STL) 3

the Standard Template LIB (STL) 4

vector

```
#include <vector>
{
  // bad:
  {
    vector<int*> iptrs(10); // 10 pointers to integer
    iptrs[5] = new int[12]; // conventional c++ allocation
    iptrs.clear(); // we just leaked 12*sizeof(int) bytes
  }

  // better:
  vector<vector<int> > iptrs; // 10 vectors of length 0
  iptrs[5].resize(12);        // 9 vectors length 0, 1 length 12
  iptrs.clear();              // we leaked nothing

  vector<double> dvec(10);
  for (int i=0; i<10; ++i) dvec[i] = 3.141*i;
  dvec.push_back(10*3.141);

  int dim = dvec.size();
  dvec.resize(dvec.size()+1);

  double* dptr = &dvec[0];    // data in vector is contigous
  MPI_Send(dptr,dvec.size(),MPI_DOUBLE,...);
}
```

– A vector can hold anything, even another templated something
– A vector calls the destructor of the data objects in the vector upon deletion, (it does NOT call delete on pointers)

comments

map is a ’pair associative container’:

```
#include <map>
#include <vector>
{
  map<int,double> data;                     // associates element ptrs with an int key
  data[5] = 3.141;                          // associates 5 with pi, no 0,1,2,3,4 exist!
  data[7] = 2.*3.141;                       // associates 7 with 2pi
  data.insert(pair<int,double>(9,2.7)); // the long way to insert something

  int dim = data.size(); // number of pairs in data

  data[5] = 2.*data[7];  // one should be certain that pairs with key 5,7 exist,
                         // otherwise the code will crash here
  data.delete(7);        // we deleted the pair with the key 7

  // Something a little bit more complex:
  map<string,vector<double> > myvectors; // vectors with a certain name
  myvectors[”solution”].resize(10);      // vector “solution” now of length 10
  myvectors[”solution”][5] = 3.141;
}
// we leaked nothing here, everything was destroyed correctly
```

– data in a map is associated with a key that can also be of any type
– data in a map is NOT contigous (its actually stored in a tree for fast access)
– how do I get my stuff out of a map if I do not know what’s exactly in there?

the Standard Template LIB (STL) 5

map comes with something called an ’iterator’

```
#include <map>
{
  map<int,double> data;                 // associates element ptrs with an int key
  data[5] = 3.141;                      // associates 5 with pi, no 0,1,2,3,4 exist!
  data[7] = 2.*3.141;                   // associates 7 with 2pi

  // now we pretend not to know what’s in data and want to print
  // everything to screen
  map<int,double>::iterator current;  // we create an iterator
  // looping through the map
  for (current=data.begin(); current!=data.end(); ++current)
  {
    // get the key and value of the current pair
    int key = current->first;
    double val = current->second;
    cout << “key “ << key << “ value “ << val << endl; // print to screen
  }

  // now we want to check whether we have something with the key ’9’
  // if so, print and then delete it
  map<int,double>::iterator haveit = data.find(9);
  if ( haveit != data.end() )
  {
    cout << “We have it, and it is” << endl
         << “key “ << haveit->first << “ value “ << haveit->second << endl;
    data.delete(haveit); // data.delete(9) would work as well
  }
}
// we leaked nothing here, everything was destroyed correctly
```

comments

the Standard Template LIB (STL) 7

if one associates something twice with the same key, it will be overwritten in a map:

```
#include <map>
#include <multimap>
{
  map<int,double> data;                    // associates element ptrs with an int key
  data[5] = 3.141;                         // associates 5 with pi, no 0,1,2,3,4 exist!
  data[7] = 2.*3.141;                      // associates 7 with 2pi

  // put something else in with the key '7'
  data[7] = 5.0;

  // now we lost our 2pi
}
```

a multimap can have more then one thing under the very same key

```
{
  multimap<int,double> data;               // associates element ptrs with an int key
  data[5] = 3.141;                         // associates 5 with pi, no 0,1,2,3,4 exist!
  data[7] = 2.*3.141;                      // associates 7 with 2pi
  // put something else in with the key '7'
  data[7] = 5.0;
  // we have two pairs with key '7' know, get them
  multimap<int,double>::iterator start = data.lower_bound(7);
  multimap<int,double>::iterator end   = data.upper_bound(7);
  multimap<int,double>::iterator current;
  for (current = start; current!= end; ++current)
    cout << “objects with 7 as key : “ << current->second << “\n”;
}
```

scary, the normal case would be to stay away from multimaps....

comments

Some of the C++ features unfamiliar to C

`cout/new/delete/string/{}/bool/namespaces/reference/files/`
`overloading`

Classes

`basic classes / public, private, protected`
`virtual classes / pure virtual classes / inheritance`
`casting`

Templates & STL

`templates / vector / map / multimap / iterators`

<– We are here now at the end of the c++ language part –>

Reference counting pointer and parameter list

`RefCountPtr / ParameterList`

Epetra parallel linear algebra objects

`Epetra_Comm / Epetra_Map / Epetra_Vector /`
`Epetra_CrsMatrix`