Classes

Classes

Records with Functions

Records were previously introduced only as a way to group data. In C++, records can also be associated with functions.

```
struct vector_t {
    unsigned  size;
    real_t *  coeffs;

    void init  ( const unsigned  n );
    void fill  ( const real_t    f );
    void scale ( const real_t    f );
};
```

For the implementation of these functions, the function name is prefixed by the record name:

```
void vector_t::init ( const uint n )
{
    ...
}
```

Classes

Records with Functions

A record function is called specifically for an instance of a record, using the same dot operator '.' as for record variables:

```
int main ()
{
    vector_t  x;

    x.init( 10 );
    x.fill( 1.0 );
    x.scale( 5.0 );

    return 0;
}
```

Classes

Record Function Implementation

Inside a record function, one has implicit access to all record variables of the specific record object, for which the function was called. Therefore, the following two functions are equivalent:

```
void
vector_t::init ( const uint n )
{
    size   = n;
    coeffs = new real_t[n];
}


...

    x.init( 10 );
```

```
void
init ( vector_t *  x,
       const uint  n )
{
    x->size   = n;
    x->coeffs = new real_t[n];
}

...

    init( x, 10 );
```

Classes

Record Function Implementation

A pointer to the corresponding record object is actually available in C++. It is called **this**. Hence, one can also write:

```
void vector_t::init ( const uint n )
{
    this->size   = n;
    this->coeffs = new real_t[n];
}
```

Record member functions can also be implemented in the definition of the **struct**. They are then automatically declared as **inline** functions:

```
struct vector_t {
    ...
    void init  ( const unsigned  n )   // inline function
    {
        size   = n;
        coeffs = new real_t[n];
    }
    ...
};
```

Classes

Records: const functions

Member functions not changing the record data should be defined as **const** functions:

```
struct vector_t {
    ...
    void    scale ( const real_t  f );
    real_t  norm2 () const;
    ...
};
```

When calling record functions for **const** objects, e.g.

```
const vector_t  x( 10 );
```

only such **const** functions are allowed to be called, since all non-**const** functions potential change the object:

```
cout << x.norm2() << endl;  // Ok: vector_t::norm2 is const
x.scale( 2 );               // ERROR: vector_t::scale is non-const
```

Classes

Records: Constructors and Destructors

There are special functions for each record:

constructors:	functions automatically called when a record type is instantiated, e.g. by **new**, and
destructor:	a function automatically called when a record variable is destroyed, e.g. by **delete**.

The name of a constructor function is identical to the name of the record, whereas the name of the destructor is the record name prefixed by '~':

```
struct vector_t {
    unsigned  size;
    real_t *  coeffs;

    vector_t  ();                      // constructor 1
    vector_t  ( const unsigned n ); // constructor 2
    ~vector_t ();                      // destructor
};
```

Classes

Records: Constructors and Destructors

By definition, constructors should create necessary resources for the object, whereas destructors should free all record object resources:

```
vector_t::vector_t ()
{
    size   = 0;
    coeffs = NULL;
}

vector_t::vector_t ( const uint  n )
{
    size   = n;
    coeffs = new real_t[n];
}
```

```
vector_t::~vector_t ()
{
    delete[] coeffs;
}
```

Remark

Constructors and the destructor have no return type. Furthermore, destructors must not have function arguments.

Classes

Records: Constructors and Destructors

Example 1: instantiated and destroyed explicitly by **new** and **delete**:

```
vector_t * x = new vector_t();       // calling constructor 1
vector_t * y = new vector_t( 10 );   // calling constructor 2

y->fill( 1.0 );

delete x;                            // calling destructor
```

Remark

If the constructor has no arguments, the corresponding parentheses can be omitted:

```
vector_t * x = new vector_t;         // calling constructor 1
```

Classes

Records: Constructors and Destructors

Example 2: instantiated and destroyed implicitly by block scope:

```
{
    vector_t  x;          // calling constructor 1
    vector_t  y( 10 );    // calling constructor 2

    y.fill( 1.0 );
}                         // destructor called automatically
```

Here, the record objects are not pointers. When leaving the block, the destructors of all record objects are called automatically! Thereby, it is ensured, that all resources are released.

Classes

Records: Special Constructors

By default, each record type has two constructors already implemented by the C++ compiler: the default constructor and the copy constructor.
The default constructor is a constructor without arguments, e.g. constructor 1 in the previous example. The copy constructor is a constructor with one argument being a constant reference to an object of the same type as the record itself:

```
struct vector_t {
    ...
    vector_t ( const vector_t & x );  // copy constructor
    ...
};
```

This is used for

```
vector_t  x( 10 );
vector_t  y( x );     // call copy constructor
vector_t  z = y;      // usually converted to z( y )
```

Classes

Records: Special Constructors

The default constructor implemented by the C++ compiler does nothing, e.g. it does not initialise the member variables. Hence, without a user implemented default constructor, the values of the member variables are random (ref. Coding Principle No. 3).
The C++ generated copy constructor simply copies the data in each member variable of the record:

```
vector_t *  x = new vector_t( 10 );
vector_t *  y = new vector_t( & x ); // now: x.coeffs == y.coeffs,
                                     // e.g. equal pointers
```

Here, changing `y` also changes `x`:

```
x->coeffs[ 5 ] = 2;   // also changes y.coeffs[ 2 ]
y->coeffs[ 3 ] = 4;   // also changes x.coeffs[ 3 ]
```

Classes

Records: Special Constructors

For vectors, instead of the pointers, the content of the array should be copied:

```
vector_t::vector_t ( const vector_t & v )
{
    size   = v.size;
    coeffs = new real_t[ size ];

    for ( uint i = 0; i < size; i++ ) coeffs[i] = v.coeffs[i];
}
```

Now, the instructions

```
vector_t *  x = new vector_t( 10 );
vector_t *  y = new vector_t( & x );

x->coeffs[ 5 ] = 2;
y->coeffs[ 3 ] = 4;
```

only effect either `x` or `y`, not both.

Records: Special Constructors

To sum this up:

Coding Principle No. 22

Always make sure, that the C++ generated default and copy constructors behave as expected. If in doubt: implement constructors by yourself.

Classes

Records: Visibility

All member variables and functions of a record were visible and accessible from any other function or datatype in C++. This makes illegal changes to the data in a record possible, e.g. change the `size` variable of `vector_t` without also changing the `coeffs` variable.

To prevent this behaviour, one can change the visibility of variables and functions using one of the following keywords:

public: variables or functions can be accessed without restrictions,

protected: variables or functions can only be accessed by member functions of the record type or by derived records (see later),

private: variables or functions can only be accessed by member functions of the record type.

Classes

Records: Visibility

Example 1:

```
struct vector_t {
private:                    // all following variables and functions
    size_t    size;         // are private
    real_t *  coeffs;

public:                     // all following variables and functions
                            // are public
    vector_t ( const size_t  n );
...
    size_t  get_size () const { return size; }
};

...

{
    vector_t  x( 10 );

    cout << x.size << endl;          // ERROR: <size> is private
    cout << x.get_size() << endl;   // Ok: <get_size> is public
}
```

Records: Visibility

Example 2:

```
struct vector_t {
private:
    size_t    size;
    real_t *  coeffs;

public:

...

protected:
    void init ( const size_t  n )
    {
        size   = n;    // Ok: <size> is visible to member functions
        coeffs = new real_t[n];
    }
};

{
    vector_t  x( 10 );

    x.init( 20 );       // ERROR: <init> is protected
}
```

Records: Visibility

Illegal states of member variables can be prevented by making them **private** and allowing modifications only via **public** member functions:

```
struct vector_t {
private:
    size_t    size;
    real_t *  coeffs;

public:
    ...
    size_t  get_size () const             { return size; }
    void    set_size ( const size_t  n ) { init( n ); }
    ...
};
```

Coding Principle No. 23

Make all member variables of a record **private** *and allow read-/write-access only via member functions.*

Classes

Records: Operator Overloading

In C++, operators, e.g. $+$, *****, $=$ or $[\cdot]$, can be defined for record datatypes to simplify access or to ensure correct behaviour.
Depending on the operator, it can be defined inside or outside the record definition, e.g. operators changing the object like $=$ or $+=$ in the record definition and binary operators working on two objects typically outside the record.
In terms of syntax, operator functions are treated like any other function. The name of the operator is defined by

> **operator** ⟨*operator name*⟩

e.g.

> **operator =**
> **operator +**
> **operator ***
> **operator []**

Records: Operator Overloading (Example)

```
struct vector_t {
    ...
    // provide index operator for vectors
    real_t  operator [] ( const size_t i ) { return coeffs[i]; }

    // arithmetics
    vector_t &  operator += ( const vector_t & v )
    {
        for ( size_t i = 0; i < size; i++ ) coeffs[i] += v.coeffs[i];
        return *this;
    }
    vector_t &  operator -= ( const vector_t & v ) { ... }
    vector_t &  operator *= ( const real_t  f )
    {
        for ( size_t i = 0; i < size; i++ ) coeffs[i] *= f;
        return *this;
    }
    ...
};

{ ...
    x += y;
    y *= 2.0;
}
```

Classes

Records: Operator Overloading

Be very careful when overloading the standard arithmetic operators, e.g. + or *, since that can lead to very inefficient code:

```
// vector addition
vector_t  operator + ( const vector_t &  v1, const vector_t &  v2 )
{
    vector_t  t( v1 );
    return ( t += v2 );
}
```

Here, a temporary object `t` has to be created. Furthermore, usually another temporary object is created by the compiler since the lifetime of `t` ends when returning from the function. Each of these temporary objects needs memory and performs a copy operation. Hence, the addition is very inefficient.

Coding Principle No. 24

Only overload operators if necessary and reasonable.

Classes

Records: Special Operators

The analog to the copy constructor is the copy operator '**=**', e.g. used in

```
x = y;
```

It is also generated by the C++ compiler by default, simply copying the individual member variables of the record. Thereby, the same problems occur, e.g. copying pointers instead of arrays.
Coding principle for copy operators (see Coding Principle No. 22):

Coding Principle No. 25

Always make sure, that the C++ generated copy operator behaves as expected. If in doubt: implement operator by yourself.

Classes

Records: Special Operators

Copy operator for vectors:

```
struct vector_t {
    ...
    vector_t &  operator = ( const vector_t &  v )
    {
        init( v.size );

        for ( uint i = 0; i < size; i++ ) coeffs[i] = v.coeffs[i];

        return *this;
    }
    ...
};
```

Now, when assigning record objects to each other, e.g.

```
x = y;
```

the user implemented copy operator is used, ensuring correctness.

Classes

Records: Special Operators

The copy operator also allows a simplified copy constructor:

```
vector_t::vector_t ( const vector_t & v )
{
    *this = v;
}
```

Classes

Classes

Records provide all mechanisms for object-oriented programming. What about classes?

C++ also provides a **class** type, e.g.

> **class** $\langle$*class name*$\rangle$ {
>
> $\vdots$
>
> };

Classes in C++ are identical to records, except for one little difference: if the visibility specifiers, e.g. **public**, **protected** and **private**, are missing, by default all member variables and functions are **private** in a class and **public** in a record.

Therefore, it is up to you, which form you prefer: classes or records.

One possible rule: for simple records without functions use a **struct**, otherwise use a **class**. But remember Coding Principle No. 22, Coding Principle No. 25 and Coding Principle No. 3 (RAII).

Classes

Application: BLAS (Version 3)

Vector type:

```
class vector_t {
private:
    size_t    size;
    real_t *  coeffs;

public:
    vector_t ( const size_t      n = 0 );
    vector_t ( const vector_t &  x );
    ~vector_t ();

    size_t  get_size () const;

    real_t   operator [] ( const size_t  i ) const;
    real_t & operator [] ( const size_t  i );

    void fill  ( const real_t  f );
    void scale ( const real_t  f );
    void add   ( const real_t  f, const vector_t & x );

    vector_t & operator = ( const vector_t & x );
};
```

Classes

Application: BLAS (Version 3)

Remarks to the vector class:

- The constructor

  ```
      vector_t ( const size_t      n = 0 );
  ```

  also serves as a default constructor since it can be called without an argument.

- The index operator

  ```
      real_t & operator [] ( const size_t  i )
      {
          return  coeffs[i];
      }
  ```

  provides write access to the coefficients since a reference to the corresponding entry is returned. Therefore, it is defined non-**const**.

Classes

Application: BLAS (Version 3)

Matrix type:

```
class matrix_t {
private:
    size_t    nrows, ncolumns;
    real_t *  coeffs;

public:
    matrix_t  ( const size_t  n, const size_t  m );
    matrix_t  ( const matrix_t &  M );
    ~matrix_t ();

    size_t  get_nrows    () const;
    size_t  get_ncolumns () const;

    real_t   operator [] ( const size_t i, const size_t j ) const;
    real_t & operator [] ( const size_t i, const size_t j );

    void fill  ( const real_t  f );
    void scale ( const real_t  f );
    void add   ( const real_t  f, const matrix_t & M );
```

Classes

Application: BLAS (Version 3)

```
    void mul_vec ( const real_t  alpha, const vector_t & x,
                   vector_t &    y ) const;

    real_t normF () const;

    matrix_t & operator = ( const matrix_t & M );

private:
    matrix_t ();
};

void
mul_mat ( ... );
```

Remarks:

- The private default constructor prevents accidental matrices of dimension 0, since it can not be called.
- The function `mul_mat` is not part of the matrix class, since the algorithm can not be connected with a specific matrix.

Application: BLAS (Version 3)

```
matrix_t * M = new matrix_t( 10, 10 );
vector_t * x = new vector_t( 10 );
vector_t * y = new vector_t( 10 );

x.fill( 1.0 );
y.fill( 0.0 );

...
M[3,4] = ... // fill matrix M
...

cout << M.normF() << endl;

M.mul_vec( -1.0, x, y );

delete x;
delete y;
delete M;
```