# Modules and Namespaces

## Header Files

Up to now, all source code was placed into one file. For reasonable programs, this is not desirable, especially if functions are reused by different programs.

Unfortunately, C++ has no real module system like, e.g. Pascal or Java, to group similar functions or types. Instead, header files are used to make C++ objects known to different source files.

As you remember, functions can be used if they were previously declared or implemented. By separating declaration and implementation into header and source file:

```
// header file: f.hh

void f ( int n, double f );
```

```
// source file: f.cc

void f ( int n, double f )
{ ... }
```

the function can be reused by just including the header file.

## Header Files

Including another file into the current source code is performed by the **include** directive:

> **#include** *“filename”* *or*
> **#include** <*filename*>

The first version is usually used for files in the same project, whereas the second version is for files from other projects, e.g. the operating system or the C++ compiler.

```
#include "f.hh"   // contains decl. of "f"

int main ()
{
    f( 42, 3.1415926 );
}
```

## Header Files

**Remark**

*By convention, the filename suffix of the header file should be either "h" (like in C), "H", "hh" or "hpp".*

# Modules and Namespaces

## C++ Library

The C++ compiler comes with a set of standard include files containing declarations of many functions:

`cstdlib`: standard C functions, e.g.

- `exit`: stop program,
- `atoi` and `atof`: string to `int` and `double` conversion,
- `qsort`: sort arrays,
- `malloc` and `free`: C-style dynamic memory management,

`cmath`: mathematical functions, e.g.

- `sqrt`: square root,
- `abs`: absolute value,
- `sin` and `cos`,
- `log`: natural logarithm

# Modules and Namespaces

## C++ Library

`cstdio`: C-style IO functions, e.g.

- `printf`: print variables to standard output,
- `fopen`, `fclose`, `fread` and `fwrite`: file IO

`cstring`: string functions, e.g.

- `strlen`: string length,
- `strcat`: string concatenation,
- `strcmp`: string comparison,
- `strcpy`: string copy

`cctype`: character tests, e.g.

- `isdigit`: test for digit,
- `islower`: test for lower case,
- `isspace`: test for white-space

etc.: `cassert`, `cerrno`, `cinttypes`, `climits`, `ctime`.

## C++ Library

Specific C++ functionality usually comes in the form of the *standard template library*. It is implemented via classes (see later) and provided by the following header files:

- `iostream`: file input/output functions and classes,
- `vector`: dynamic containers similar to arrays,
- `valarray`: similar to `vector` but better suited for numerics,
- `limits`: functions for determining type limits, e.g. minimal or maximal values,
- `map`: associative array, e.g. indices are arbitrary types,
- `list`: provides standard list and iterators,
- `complex`: provides complex datatype,
- etc.

The specific classes and their usage will be discussed later.

## Libraries without Headers (LAPACK)

LAPACK is written in Fortran and no header files exist for C++. Therefore, we will have to write them ourselves. Consider

```
 SUBROUTINE DGESVD( JOBU, JOBVT, M, N, A, LDA, S, U, LDU, VT, LDVT,
$                   WORK, LWORK, INFO )

 CHARACTER          JOBU, JOBVT
 INTEGER            INFO, LDA, LDU, LDVT, LWORK, M, N
 DOUBLE PRECISION   A( LDA, * ), S( * ), U( LDU, * ),
$                   VT( LDVT, * ), WORK( * )
```

To define a C++ function corresponding to the above Fortran function the datatypes have to be mapped to C++ types. In Fortran, every variable is provided as a pointer, hence:

$$
\begin{aligned}
&\texttt{CHARACTER} && \rightarrow \texttt{char *}, \\
&\texttt{INTEGER} && \rightarrow \texttt{int *} \qquad \text{and} \\
&\texttt{DOUBLE PRECISION} && \rightarrow \texttt{double *}.
\end{aligned}
$$

## Libraries without Headers (LAPACK)

Fortran function names are in lower case and end with and underscore '_', when seen from C or C++. Hence, the name of the above Fortran function is `dgesvd_`:

```
void dgesvd_ ( char * jobu, char * jobv, int * n, int * m,
               double * A, int * lda, double * S, double * U,
               int * ldu, double * V, int * ldv, double * work,
               int * lwork, int * info );
```

Furthermore, there is a difference between C and C++ functions. Fortran only provides C-style functions, whereas the above is a C++ function. To tell the compiler, that a C-style function should be declared, the **extern "C"** instruction is provided:

```
extern "C" {
void dgesvd_ ( char * jobu, char * jobv, int * n, int * m,
               double * A, int * lda, double * S, double * U,
               int * ldu, double * V, int * ldv, double * work,
               int * lwork, int * info );
}
```

# Modules and Namespaces

## Libraries without Headers (LAPACK)

Afterwards, the function `dgesvd_` can be used like any other C++ function. To compute the SVD $M = U \cdot S \cdot V^T$ of a matrix $M$, the code looks like:

```
int        n     = 10;
double *   M     = new double[ n*n ];
char       jobu  = 'O';                  // overwrite M with U
char       jobv  = 'S';                  // store V^T in VT
int        info  = 0;
int        lwork = 10*n*n;
double *   work  = new double[ work ]; // workspace for dgesvd
double *   S     = new double[ n ];
double *   VT    = new double[ n*n ];

... // fill M

dgesvd_( & jobu, & jobv, & n, & n, M, & n, S, M, & n, V, & ldv,
         work, & lwork, & info );
```

## Header File Recursion

The **include** directive can be seen as simple text replacement: the directive is replaced by the content of the corresponding file.

```
#include "f.hh"

int main ()
{
    f( 42, 3.1415926 );
}
```

```
void f ( int n, double f );

int main ()
{
    f( 42, 3.1415926 );
}
```

This might lead to infinite loops, if you have recursive **include** directives in different files, e.g. “file1.hh” includes “file2.hh”, which by itself includes “file1.hh”.

```
// FILE: file1.hh

#include "file2.hh"

...
```

```
// FILE: file2.hh

#include "file1.hh"

...
```

## Header File Encapsulation

To prevent infinite loops, two other directives are provided:

> **#ifndef** ⟨*NAME*⟩
> ⋮
> **#endif**

tests, if the symbol ⟨NAME⟩ was previously defined by the directive

> **#define** ⟨*NAME*⟩

If it was not defined, all source code between the **ifndef** directive and the corresponding **endif** will be included by the C++ compiler. Otherwise, the source code will be omitted.

**Remark**

> *It is recommended to name the symbol ⟨NAME⟩ after the name of the header file.*

## Header File Encapsulation

Now, for the recursive example:

```cpp
// FILE: file1.hh

#ifndef __FILE1_HH
#define __FILE1_HH

#include "file2.hh"

#endif
```

```cpp
// FILE: file2.hh

#ifndef __FILE2_HH
#define __FILE2_HH

#include "file1.hh"

#endif
```

If “file1.hh” is included by a source file, the symbol “__FILE1_HH” will be defined and the content of the header file included. Similar, `#include ‘‘file2.hh’’` will be replaced by the content of “file2.hh”. If now again “file1.hh” should be included, “__FILE1_HH” is already defined and the content of “file1.hh” is omitted, stopping the recursion.

## Header File Encapsulation

**Coding Principle No. 19**

*Always encapsulate your header files by an* **ifndef-define-endif** *construct.*

## Inline Functions

It is also possible to implement a function in a header file. In that case, it has to be declared **inline**, because otherwise, the function is defined in each source file, where the header is included. If you than compile all source files together, you would have multiple instances of the same function, which is not allowed.

```
#ifndef __SQUARE_HH
#define __SQUARE_HH

inline
double
square ( const double x )
{
    return x*x;
}

#endif
```

## Variables

Beside functions, you can also declare variables in header files. For non-const data, the declaration and definition has to be separated to prevent multiple instances. In the header file, the variables have to be declared with the keyword **extern**:

```
// header file: real.hh

#ifndef  __REAL_HH
#define  __REAL_HH

typedef double  real_t;

const  real_t  PI = 3.1415926;   // const: "extern" not needed
extern real_t  eps;
extern int     stepwidth;

#endif
```

## Variables

The definition than has to be made in a source file:

```
// source file: real.cc

#include "real.hh"  // for real_t

real_t  eps       = 1e-8;
int     stepwidth = 1024;
```

Afterwards, every module including the corresponding headerfile has access to the variables `eps` and `stepwidth`:

```
#include "real_t.hh"

int
main ()
{
    eps = 1e-4;
    cout << stepwidth << endl;

    return 0;
}
```

## Module Scope

If a function or variable is declared in a header file, it is globally visible by all other parts of the program. It is in global scope.

For variables, that simplifies access, e.g. global variables do not need to be supplied as function parameters. But it has a major drawback: every function can change the variable, independent on possible side effects.

Better approach: define a function for changing the variable. That way, the access can be controlled:

```
// header
void
set_eps ( const real_t  aeps );

real_t
get_eps ();
```

```
// source
static real_t eps = 1e-8;

void
set_eps ( const real_t  aeps )
{
  if      ( aeps < 0.0 ) eps = 0.0;
  else if ( aeps > 1.0 ) eps = 1.0;
  else                   eps = aeps;
}
```

# Modules and Namespaces

## Module Scope

Therefore:

**Coding Principle No. 20**

*Only if absolutely neccessary make non-const variables global.*

**Remark**

*The* **static** *declaration of a variable or function in a source file prevents other modules from using that variable or function, respectively.*

## Namespaces

Following situation: we have written datatypes and functions for dense matrices in a module, e.g.

```
struct matrix_t {
    size_t    nrows, ncolumns;
    real_t *  coeffs;
};

matrix_t *  init    ( ... );
void        mul_vec ( ... );
```

and you want to join that with another module for sparse matrices:

```
struct matrix_t {
    size_t    nrows, ncolumns, nnzero;
    size_t *  rowptr, * colind;
    real_t *  coeffs;
};

matrix_t * init    ( ... );
void       mul_vec ( ... );
```

# Modules and Namespaces

## Namespaces

Problem: although functions with the same name are allowed, two datatypes must not have the same name.

Solution 1: rename all occurences of `matrix_t` for sparse matrices, and change all functions, or

Solution 2: put each type and function set into a different namespace.

A namespace is a mechanism in C++ to group types, variables and functions, thereby defining the scope of these objects, similar to a block. Till now, all objects were in the global namespace.

Namespace definition:

$$\textbf{namespace}\ \langle \textit{namespace name} \rangle\ \{$$
$$\vdots$$
$$\}$$

## Namespaces

Applied to the two matrix modules from above:

```
namespace Dense {

struct matrix_t {
  unsigned  nrows, ncolumns;
  real_t *  coeffs;
};


matrix_t *  init    ( ... );
void        mul_vec ( ... );

}
```

```
namespace Sparse {

struct matrix_t {
  unsigned    nrows, ncolumns, nnzero;
  unsigned *  rowptr, * colind;
  real_t *    coeffs;
};

matrix_t * init    ( ... );
void       mul_vec ( ... );

}
```

This defines two namespaces `Dense` and `Sparse`, each with a definition of `matrix_t` and corresponding functions.

## Namespace Access

The access to functions or types in a namespace is performed with the namespace operator "::" :

```
Dense::matrix_t *   D = Dense::init(  10, 10 );
Sparse::matrix_t *  S = Sparse::init( 10, 10, 28 );

Dense::mul_vec(  1.0, D, x, y );
Sparse::mul_vec( 1.0, S, x, y );
```

## Namespace Access

To make all objects in a namespace visible to the local namespace, the keyword **using** is provided:

```
using namespace Dense;
using namespace Sparse;

Dense::matrix_t *   D = init( 10, 10 );     // call to Dense::init
Sparse::matrix_t *  S = init( 10, 10, 28 ); // call to Sparse::init

mul_vec( 1.0, D, x, y );                    // call to Dense::mul_vec
mul_vec( 1.0, S, x, y );                    // call to Sparse::mul_vec
```

**Remark**

*Remember, that types must have different names.*
*Hence, the types for `D` and `S` have to be named with their namespaces.*

## Namespace Access

Restrict the usage of **using** to source files and avoid **using** directives in header files, because all modules including the header would also include the corresponding **using** instruction:

```
// header file: vector.hh

#include "dense.hh"

using namespace Dense;

...  // vector definitions
```

```
// source file: module.cc

#include "vector.hh"
#include "sparse.hh"

using namespace Sparse;

void  f ( matrix_t & M );
```

Here, `matrix_t` is ambiguous, e.g. either `Dense::matrix_t` or `Sparse::matrix_t`.

## Namespace Aliases

It is also possible to define an alias for a namespace, e.g. to abbreviate it:

```
namespace De = namespace Dense;
namespace Sp = namespace Sparse;

De::matrix_t *  D = De::init( 10, 10 );
Sp::matrix_t *  S = Sp::init( 10, 10, 28 );

De::mul_vec( 1.0, D, x, y );
Sp::mul_vec( 1.0, S, x, y );
```

# Modules and Namespaces

## Nested Namespaces

Namespaces can also be nested and different parts of a namespace can be defined in different modules:

```
namespace LinAlg {
    namespace Dense {
        ...
    }
}
```

```
namespace LinAlg {
    namespace Sparse {
        ...
    }
}
```

```
LinAlg::Dense::matrix_t *  D = LinAlg::Dense::init(  10, 10 );
LinAlg::Sparse::matrix_t * S = LinAlg::Sparse::init( 10, 10, 28 );

LinAlg::Dense::mul_vec(  1.0, D, x, y );
LinAlg::Sparse::mul_vec( 1.0, S, x, y );
```

## Anonymous Namespaces

Namespaces can also be defined without a name:

```
namespace {
    void f ()
    {
        ...
    }
}
```

The C++ compiler will then automatically assign a unique, hidden name for such a namespace. This name will be different in different modules.

Functions in an anonymous namespace can be used without specifying their namespace name:

```
namespace {
    void f () { ... }
}

void g () { f(); }
```

## Anonymous Namespaces

On the other hand, since the automatically assigned name is unique per module, only functions in the same module as the anonymous namespace can access the functions within:

```
// module 1

namespace {
    void f () { ... }
}

void g ()
{
    f(); // Ok: same module
}
```

```
// module 2

void h ()
{
    f(); // Error: unknown
         // function "f"
}
```

Such functions are therefore hidden for other modules and purely local to the corresponding module.

## Anonymous Namespaces

**Remark**

*If an anonymous namespace is defined in a header file, each module including the header would define a new, local namespace!*

**Coding Principle No. 21**

*Put module local functions into an anonymous namespace.*

This approach is different from the previous C-style version using **static** functions and variables and should be preferred.

## The std Namespace

All functions (and classes) of the C++ standard library, e.g. `sqrt` or `strcpy` are part of the std namespace. Previous use always assumed a corresponding **using** command:

```
using namespace std;

const real_t  PI     = 3.14159265358979323846;
const real_t  sqrtPI = sqrt( PI );

cout << sqrtPI << endl;
```

is equivalent to

```
const real_t  PI     = 3.14159265358979323846;
const real_t  sqrtPI = std::sqrt( PI );

std::cout << sqrtPI << std::endl;
```