Advanced Datatypes

Type Definition

Often you do not always want to care about the actual datatype used in your program, e.g. if it is `float` or `double` or if strings are `char *`, but instead give the types more reasonable names, e.g. `real` or `string`. In C++ you can do this via **typedef**:

> **typedef** ⟨*data type*⟩ ⟨*name*⟩*;*

Afterwards, ⟨name⟩ can be used like any other datatype:

```
typedef  double     real_t;
typedef  char *     string_t;
typedef  real_t **  matrix_t;  // pointers of pointers

const string_t  str = "String";
matrix_t        A   = new real_t*[ 10 ];
real_t          f   = real_t( 3.1415926 );
```

Type Definition

Remark

A `real_t` datatype allows you to easily change between `float` and `double` in your program.

To simplify the destinction between variables and datatypes, the following is strongly advised:

Coding Principle No. 18

Follow a strict convention in naming new types, e.g. with special prefix or suffix.

Advanced Datatypes

Predefined Types

The C++ library and the operating system usually define some abbreviations for often used types, e.g.

- `uint`: unsigned integer, sometimes special versions `uint8`, `uint16` and `uint32` for 8, 16 and 32 bit respectively,
- similar `int8`, `int16` and `int32` are defined for signed integers,
- `size_t` : unsigned integer type for holding size informations best suited for the local hardware
- `ssize_t` : analog to `size_t` but signed integer (not always available)

Records

Working with vectors and matrices always involved several variables, e.g. the size and the arrays. That results in many arguments to functions and hence, to possible errors. It would be much better to store all associated data together. That is done with records:

> **struct** ⟨*record name*⟩ {
>   ⟨*datatype 1*⟩ ⟨*name 1*⟩;
>   ⋮
>   ⟨*datatype n*⟩ ⟨*name n*⟩;
> };

By defining a **struct**, also a new type named ⟨record name⟩ is defined.

Records

Example:

```
struct vector_t {
    size_t    size;
    real_t *  coeffs;
};

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

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

struct triangle_t {
    int     vtx_idx[3];  // indices to a vertex array
    real_t  normal[3];
    real_t  area;
};
```

Access Records

The individual variables in a record are accessed via “.”, e.g.:

```
vector_t  x;

x.size   = 10;
x.coeffs = new real_t[ x.size ];
```

If a pointer to a record is given, the access can be simplified. Instead of “*” (dereference) and “.”, the operator “->” is provided:

```
vector_t *  x = new vector_t;

x->size = 10;
x->data = new real_t[ x->size ];

cout << (*x).size << endl; // alternative
```

Advanced Datatypes

Access Records

In case of a reference, e.g. `vector_t &`, the standard access has to be used, e.g. via "`.`":

```
vector_t  x;

x.size   = 10;
x.coeffs = new real_t[ x.size ];

vector_t & y = x;

cout << y.size << endl;
cout << y.coeffs[5] << endl;
```

Records and Functions

Records can be supplied as normal function arguments, either as call-by-value or call-by-reference:

```
double dot  ( const vector_t  x, const vector_t    y );
void   fill ( const real_t    f, vector_t &        y );
void   add  ( const real_t    f, const vector_t &  x,
              vector_t &      y );
```

When using call-by-value, a copy of the complete record is actually created. For large records, this can be a significant overhead:

```
struct quadrule_t {
    real_t   points[  100 ];
    real_t   weights[ 100 ];
};

double quadrature ( const quadrule_t  rule,
                    double ( * func ) ( const double x ) );
```

Records and Functions

In such cases, call-by-reference with a **const** argument is necessary to avoid this overhead:

```
double quadrature ( const quadrule_t &  rule,
                    double ( * func ) ( const double x ) );
```

Here, only a single pointer is supplied to the function instead of 200 `real_t` values.

Application: BLAS (Version 2)

Modified BLAS function set using the previous record types for vectors and matrices:

```
inline vector_t *
vector_init ( const unsigned  i )
{
    vector_t *  v = new vector_t;

    v->size = i;
    v->coeffs = new real_t[ i ];

    for ( unsigned i = 0; i < n; i++ )  // RAII
        v->coeffs[i] = 0.0;

    return v;
}

inline matrix_t *
matrix_init ( const unsigned  n, const unsigned  m )
{ ... }
```

Application: BLAS (Version 2)

```
// vector functions
void fill  ( const double f, vector_t & x );
void scale ( const double f, vector_t & x );

void add   ( const double f, const vector_t & x, vector_t & y )
{
    for ( unsigned i = 0; i < n; i++ )
        y.coeffs[i] += f * x.coeffs[i];
}

double        dot   ( const vector_t & x, const vector_t & y );
inline double norm2 ( const vector_t & x )
{ return sqrt( dot( x, x ) ); }

// matrix functions
void fill  ( const double f, matrix_t & M );
void scale ( const double f, matrix_t & M );
void add   ( const double f, const matrix_t & A, matrix_t & M );

inline double
normF ( double & M )
{ ... }  // can not use vector based norm2!
```

Application: BLAS (Version 2)

```
void
mul_vec ( const double       alpha,
          const matrix_t &   M,
          const vector_t &   x,
          vector_t &         y )
{
    for ( unsigned i = 0; i < M.nrows; i++ )
    {
        double  f = 0.0;

        for ( unsigned j = 0; j < M.ncolumns; j++ )
            f += get_entry( M, i, j ) * x.coeffs[j];

        y.coeffs[i] += alpha * f;
    }
}

void
mul_mat ( const double       alpha,
          const matrix_t &   A,
          const matrix_t &   B,
          matrix_t &         C );
```

Application: BLAS (Version 2)

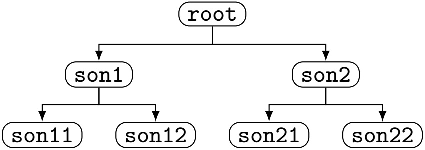
```
matrix_t * M = matrix_init( 10, 10 );
vector_t * x = vector_init( 10 );
vector_t * y = vector_init( 10 );

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

...  // fill matrix M

cout << normF( M ) << endl;

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

Recursive Records

Records can have variables of it's own type in the form of a pointer. That way, recursive structures can be defined, e.g. a binary tree:

```
struct node_t {
    int       val;
    node_t *  son1, * son2;
};

node_t  root;
node_t  son1, son2;
node_t  son11, son12, son21, son22;

root.son1 = & son1;  root.son2 = & son2;
son1.son1 = & son11; son1.son2 = & son12;
son2.son1 = & son21; son2.son2 = & son22;
```

The above code yields:



Recursive Records

Insert new value in binary tree:

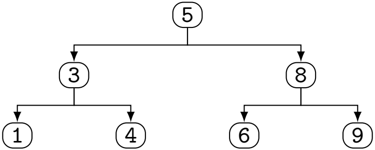
```
void
insert ( const node_t & root, const int val )
{
    if ( val < root.val )
    {
        if ( root.son1 != NULL )
            insert( * root.son1, val );
        else
        {
            root.son1        = new node_t;
            root.son1->val   = val;
            root.son1->son1 = NULL;
            root.son1->son2 = NULL;
        }
    }
    else
    {
        if ( root.son2 != NULL )
            insert( * root.son2, val );
        else
            ...
}   }
```

Recursive Records

Example for insertion:

```
int     values[7] = { 5, 3, 1, 4, 8, 6, 9 };
node_t  root;

root.son1 = root.son2 = NULL;
root.val  = values[0];

for ( int i = 1; i < 7; i++ )
    insert( root, values[i] );
```

yields:



Recursive Records

Looking for value in binary tree:

```
bool
is_in ( const node_t & root, const int val )
{
    if ( root.val == val )
        return true;

    return is_in( * root.son1, val ) ||
           is_in( * root.son2, val );
}

...

    cout << is_in( root, 6 ) endl;  // yields true
    cout << is_in( root, 7 ) endl;  // yields false
```

Arrays of Records

Like any other datatype, records can also be allocated in the form of an array:

```
struct coord_t {
    real_t  x, y, z;
};

coord_t   coordinates[ 10 ];
```

for fixed sized array or

```
coord_t * coordinates = new coord_t[ 10 ];
```

using dynamic memory management.

Arrays of Records

The access to record variables then comes after addressing the array entry:

```
for ( unsigned i = 0; i < 10; i++ )
{
    coordinates[i].x = cos( real_t(i) * 36.0 * pi / 180.0 );
    coordinates[i].y = sin( real_t(i) * 36.0 * pi / 180.0 );
    coordinates[i].z = real_t(i) / 10.0;
}
```

If instead, an array of pointers to a record is allocated:

```
coord_t ** coordinates = new coord_t*[ 10 ];

for ( int i = 0; i < 10; i++ )
    coordinates[i] = new coord_t;
```

the access if performed with the arrow operator ->:

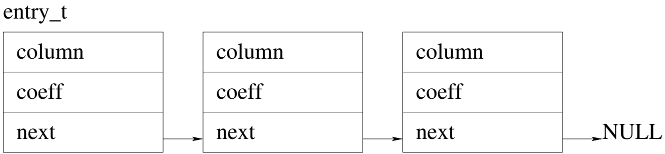
```
    coordinates[i]->x = cos( real_t(i) * 36.0 * pi / 180.0 );
```

Record Application: Sparse Matrices

We only want to store nonzero entries in a sparse matrix. For this, each entry is stored in a record type, containing the column index and the coefficient:

```
struct entry_t {
    unsigned   column;  // column of the entry
    real_t     coeff;   // actual coefficient
    entry_t *  next;    // next entry in row
};
```

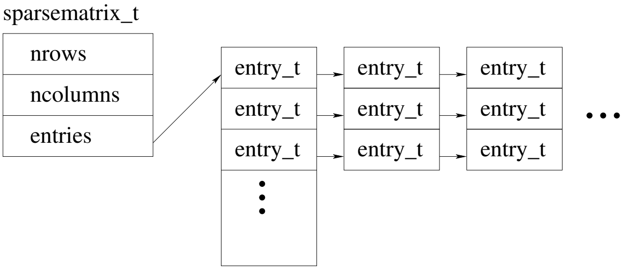
All entries in a row are stored in a list, provided by the `next` pointer in an entry type. A **NULL** value of `next` signals the end of the list.



Record Application: Sparse Matrices

A sparse matrix is then allocated as an array of entry lists per row:

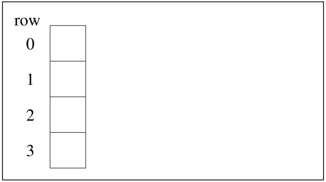
```
struct sparsematrix_t {
    unsigned   nrows, ncolumns;
    entry_t *  entries;
};
```



Record Application: Sparse Matrices

As an example, consider the matrix

$$\begin{pmatrix} 1 & & 3 & \\ & 2 & & -1 \\ -4 & -1 & 1 & \\ 1 & & & 3 \end{pmatrix}$$

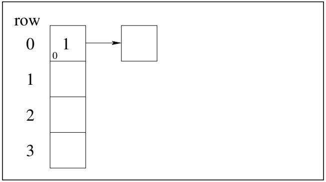



```
sparsematrix_t  S;
entry_t *       entry;

S.nrows = 4; S.ncolumns = 4;
S.entries = new entry_t[4];

// first row
entry = & S.entry[0];
entry->column = 0;  entry->coeff = 1.0; entry->next = new entry_t;

entry = entry->next;
entry->column = 2;  entry->coeff = 3.0; entry->next = NULL;
...
```

Record Application: Sparse Matrices

As an example, consider the matrix

$$\begin{pmatrix} 1 & & 3 & \\ & 2 & & -1 \\ -4 & -1 & 1 & \\ 1 & & & 3 \end{pmatrix}$$

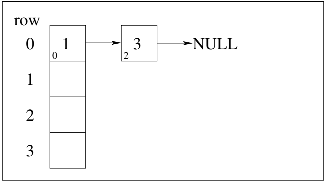



```
sparsematrix_t  S;
entry_t *       entry;

S.nrows = 4; S.ncolumns = 4;
S.entries = new entry_t[4];

// first row
entry = & S.entry[0];
entry->column = 0;  entry->coeff = 1.0; entry->next = new entry_t;

entry = entry->next;
entry->column = 2;  entry->coeff = 3.0; entry->next = NULL;
...
```

Record Application: Sparse Matrices

As an example, consider the matrix

$$\begin{pmatrix} 1 & & 3 & \\ & 2 & & -1 \\ -4 & -1 & 1 & \\ 1 & & & 3 \end{pmatrix}$$

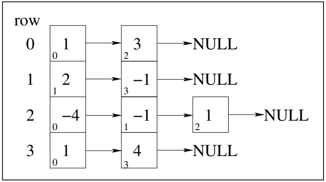



```
sparsematrix_t  S;
entry_t *       entry;

S.nrows = 4; S.ncolumns = 4;
S.entries = new entry_t[4];

// first row
entry = & S.entry[0];
entry->column = 0;  entry->coeff = 1.0; entry->next = new entry_t;

entry = entry->next;
entry->column = 2;  entry->coeff = 3.0; entry->next = NULL;
...
```

Record Application: Sparse Matrices

As an example, consider the matrix

$$\begin{pmatrix} 1 & & 3 & \\ & 2 & & -1 \\ -4 & -1 & 1 & \\ 1 & & & 3 \end{pmatrix}$$




```
sparsematrix_t  S;
entry_t *       entry;

S.nrows = 4; S.ncolumns = 4;
S.entries = new entry_t[4];

// first row
entry = & S.entry[0];
entry->column = 0;  entry->coeff = 1.0; entry->next = new entry_t;

entry = entry->next;
entry->column = 2;  entry->coeff = 3.0; entry->next = NULL;
...
```

Record Application: Sparse Matrices

Matrix-Vector multiplication:

```
void
mul_vec ( const real_t   alpha, const sparsematrix_t & S,
          const vector_t x, vector_t & y )
{
    for ( unsigned i = 0; i < S.nrows; i++ )
    {
        real_t    f     = 0.0;
        entry_t * entry = & S.entries[i];

        while ( entry != NULL )
        {
            f      += entry->coeff * x[ entry->column ];
            entry  = entry->next;
        }

        y[ i ] += alpha * f;
    }
}
```

Record Application: Sparse Matrices (Version 2)

We can store sparse matrices even more memory efficient, without pointers. For this, we'll use three arrays:

- `colind`: stores column indices for all entries, sorted by row,
- `coeffs`: stores all coefficients in same order as in `colind` and
- `rowptr`: stores at `rowind[i]` the position of the first values corresponding to row i in the arrays `colind` and `coeffs`. The last field, contains the number of nonzero entries.

This format is known as the *compressed row storage* format.

```
struct crsmatrix_t {
    unsigned    nrows, ncolumns;
    unsigned *  rowptr;
    unsigned *  colind;
    real_t *    coeffs;
};
```

Record Application: Sparse Matrices (Version 2)

For the matrix

$$\begin{pmatrix} 1 & & 3 & \\ & 2 & & -1 \\ -4 & -1 & 1 & \\ 1 & & & 3 \end{pmatrix}$$

the corresponding source code is:

```
crsmatrix_t  S;
unsigned     rowptr[] = { 0, 2, 4, 7, 9 };
unsigned     colind[] = { 0, 2,   1,  3,    0,  1, 2,   0, 3 };
real_t       coeffs[] = { 1, 3,   2, -1,   -4, -1, 1,   1, 3 };

S.nrows = 4; S.ncolumns = 4;
S.rowptr = rowptr;
S.colind = colind;
S.coeffs = coeffs;
```

Record Application: Sparse Matrices (Version 2)

Matrix-Vector multiplication:

```
void
mul_vec ( const real_t   alpha, const crsmatrix_t & S,
          const vector_t x, vector_t & y )
{
    for ( unsigned row = 0; row < S.nrows; row++ )
    {
        real_t         f  = 0.0;
        const unsigned lb = S.rowptr[ row ];
        const unsigned ub = S.rowptr[ row+1 ];

        for ( unsigned j = lb; j < ub; j++ )
            f += S.coeffs[ j ] * x[ S.colind[ j ] ];

        y[ i ] += alpha * f;
    }
}
```

Enumerations

A special datatype is available to define enumerations:

> **enum** ⟨*enum name*⟩ {
>
> ⟨*name 1*⟩, ⟨*name 2*⟩, ..., ⟨*name n*⟩
>
> };

Example:

```
enum matrix_type_t { unsymmetric, symmetric, hermitian };

matrix_type_t  t;

if ( t == symmetric ) { ... }
```

Enumerations are handled as integer datatypes by C++. By default, the members of an enumeration are numbered from 0 to $n-1$, e.g. <name 1>$= 0$, <name 2>$= 1$, etc..

Enumerations

One can also define the value of the enumeration members explicitly:

```
enum matrix_size_t { small = 10, middle = 1000, large = 1000000 };

if      ( nrows < small  ) { ... }
else if ( nrows < middle ) { ... }
else if ( nrows < large  ) { ... }
else                       { ... }
```

Since enumerations are equivalent to integer types, they can also be used in **switch** statements:

```
switch ( type )
{
case symmetric:   ...; break;
case hermitian:   ...; break;

case unsymmetric:
default:          ...;
}
```

Unions

A union is a special record datatype where all variables share the same memory, i.e. changing one variable changes all other variables.

$$\textbf{union}\ \langle \textit{union name} \rangle\ \{$$
$$\quad \langle \textit{datatype 1} \rangle\ \langle \textit{name 1} \rangle;$$
$$\quad \vdots$$
$$\quad \langle \textit{datatype n} \rangle\ \langle \textit{name n} \rangle;$$
$$\};$$

Example:

```
union utype_t {
    int    n1;
    int    n2;
    float  f;
};
```

```
utype_t  u;

u.n1 = 2;
cout << u.n2 << endl;  // yields 2
cout << u.f  << endl;  // ???
```

Unions

Unions can also be used inside records:

```
enum smat_type_t { ptr_based, crs_based };

struct general_sparse_matrix_t {
    smat_type_t  type;

    union {
        sparsematrix_t  ptrmat;
        crsmatrix_t     crsmat;
    } matrix;
};

general_sparse_matrix_t  S;

S.type = ptr_based;
S.matrix.ptrmat.nrows = 10;
```

Remark

Typical usage for unions: save memory for different representations.

Unions

The name `matrix` of the **union** can be omitted. The access is then as if it were a direct member of the **struct**.

```
enum smat_type_t { ptr_based, crs_based };

struct general_sparse_matrix_t {
    smat_type_t  type;

    union {
        sparsematrix_t  ptrmat;
        crsmatrix_t     crsmat;
    };
};

general_sparse_matrix_t  S;

S.type = ptr_based;
S.ptrmat.nrows = 10;
```