SymbolicC++ introduces, amongst others, the `Symbolic` class which is used for all symbolic computation. The `Symbolic` class provides almost all of the features required for symbolic computation including symbolic terms, substitution, non-commutative multiplication and vectors and matrices.

All the necessary classes and definitions are obtained by

```
#include "symbolicc++.h"
```

int the C++ source file.

There are a number of constructors available for `Symbolic`:

```
Symbolic zero;                              // default value is 0
Symbolic int_one1(1), int_one2 = 1;         // construction from int
Symbolic dbl_one1(1.0), dbl_one2 = 1.0;     // construction from double
Symbolic half = Symbolic(1)/2;              // fraction 1/2
Symbolic a("a");                            // symbol a
Symbolic b("b", 3);                         // vector (b0, b1, b2)
Symbolic c = b(2);                          // copy constructor, c = b2
Symbolic A("A", 2, 3);                      // matrix A with 2 rows and 3 columns
Symbolic d = A(1, 2);                       // copy constructor, d = A(1, 2);
Symbolic e = (a, c, d);                     // vector (a, b2, A(1, 2))
Symbolic B = ( ( half,      a ),            // matrix B = [  1/2     a    ]
               ( c,    A(0,0) ) );          //             [   b2   A(0,0) ]
```

The `,` operator has been overloaded to create lists of type STL `list<Symbolic>` which can be assigned to `Symbolic` to create vectors and matrices as shown for `v` and `B`. Matrices and vectors are indexed using the `()` and `(,)` operators.

All the standard arithmetic operators are provided for `Symbolic` as well as the usual functions `cos`, `sin`, `exp`, `tan`, `cot`, `sec`, `csc`, `sinh`, `cosh`, `ln`, `pow` or alternatively `(x^y)`, and `sqrt`. The precedence of `^` is lower than `|+|`so the parenthesis `(x^y)` are usually necessary.

Symbolic C++ also includes an `Equation` class for expressing equality (or substitution) usually constructed using the `==` operator:

```
Equation eq = (a == a*c - d);
```

`Equations` also serve as logical variables, in the sense that they can be cast to `bool`:

```
if(eq) cout << "a == a*c - d" << endl;   // a != a*c - d
else   cout << "a != a*c - d" << endl;
if(a == a)                               // creates the equation (a == a)
 cout << "a is a" << endl;               // the if statement implicitly
                                         // casts the equation to bool
```

Symbols can depend on eachother using the [] operator:

```
Symbolic x("x"), y("y"), t("t");
cout << y << endl;                       // independent y
cout << y[x] << endl;                    // y[x] (y dependent on x, explicit)
cout << y << endl;                       // independent y
cout << y[x,t] << endl;                  // y[x,t] (y dependent on x and t)
cout << y << endl;                       // independent y
x = x[t];                                // x depends on t (implicit)
y = y[x];                                // y depends on x
cout << y << endl;                       // y[x[t]]
```

Substitution is specified via equations and the [] operator:

```
Symbolic v("v");
Symbolic u = (v^5) + cos(v-2);           // u depends implicitly on v
cout << u[v == 2] << endl;               // 33
cout << u[cos(v-2) == sin(v-2), v == 2]  // 32
     << endl;
cout << u[v == 2, cos(v-2) == sin(v-2)]  // 33
     << endl;
cout << u.subst(v == 2) << endl;         // 33
cout << u.subst_all(v == 2) << endl;     // 33
cout << u.subst(v == v*v) << endl;       // v^10 + cos(v^2 - 2)
cout << u.subst_all(v == v*v) << endl;   // never returns
```

The above example demonstrates that substitution proceeds from left to right. The member function subst can also be used for substitution, as well as subst_all. The difference between the two methods is that subst substitutes in each component of an expression only once while subst_all attempts to perform the substitution until the substitution fails, thus for v → v*v we have the never ending substitution sequence v → v^2 → v^4 → v^8 → · · ·.

Symbolic variables can be either commutative or non-commutative. By default symbolic variables are commutative, commutativity is toggled using the ~ operator:

```
Symbolic P("P"), Q("Q");
cout << P*Q - Q*P << endl;                 // 0
cout << ~P*~Q - ~Q*~P << endl;             // P*Q - Q*P
cout << P*Q - Q*P << endl;                 // 0
P = ~P;                                    // P is non-commutative
cout << P*Q - Q*P << endl;                 // 0
Q = ~Q;                                    // Q is non-commutative
cout << P*Q - Q*P << endl;                 // P*Q - Q*P
cout << (P*Q - Q*P)[Q == ~Q] << endl;      // 0
cout << P*Q - Q*P << endl;                 // P*Q - Q*P
Q = ~Q;                                    // Q is commutative
cout << P*Q - Q*P << endl;                 // 0
```

It is also possible to determine the coefficient of expressions using the method `coeff`, and additional power can be specified:

```
Symbolic m("m"), n("n");
Symbolic result = (2*m - 2*n)^2;           // 4*(m^2) - 8*m*n + 4*(n^2)
cout << result.coeff(m^2) << endl;         // 4
cout << result.coeff(n,2) << endl;         // 4
cout << result.coeff(m) << endl;           // -8*n
cout << result.coeff(m*n) << endl;         // -8
cout << result.coeff(m,0) << endl;         // constant term: 4*(n^2)
cout << result.coeff(m^2,0) << endl;       // constant term: -8*m*n + 4*(n^2)
cout << result.coeff(m*n,0) << endl;       // constant term: 4*(m^2) + 4*(n^2)
```

Differentiation and elementary intergration is supported via the functions `df` and `integrate`:

```
Symbolic p("p"), q("q");
cout << df(p, q) << endl;                  // 0
cout << df(p[q], q) << endl;               // df(p[q], q)
cout << df(p[q], q, 2) << endl;            // df(p[q], q, q) (2nd derivative)
cout << df(cos(p[q]^2) - (q^2)*sin(q),q) // -2*sin(p[q]^(2))*p[q]*df(p[q],q)
     << endl;                              //  - 2*q*sin(q)-q^(2)*cos(q)
cout << integrate(p, q) << endl;           // p*q
cout << integrate(p[q], q) << endl;        // integrate(p[q], q)
cout << integrate(ln(q), q) << endl;       // q*ln(q) - q
```

A number of operattions are defined on `Symbolic` which are dependent on the underlying value. For example, a symbolic expression which evaluates to an integer can be cast to `int` and similarly for `double`. Note that `double` is never simplified to `int`, for example $2.0 \not\to 2$ while fractions do $\frac{2}{2} \to 1$.

```
Symbolic z("z");
cout << int(((z-2)^2) - z*(z-4))         // 4
     << endl;
cout << int(((z-2)^2) - z*(z-4.0))       // 4
     << endl;
cout << int(((z-2.0)^2) - z*(z+4))       // error: -8*z
     << endl;
cout << int(((z-2.0)^2) - z*(z-4))       // error: 4.0 is not an integer
     << endl;
cout << double(((z-2.0)^2) - z*(z-4))    // 4.0
     << endl;
```

The matrix operations `det` and `tr`, scalar product `a|b`, cross product `%` and methods `rows`, `columns`, `row column`, `identity`, `transpose`, `vec`, `kron dsum` and `inverse` are only defined on matrices with appropriate properties.

```
Symbolic X("X", 3, 3), Y("Y", 3, 3);
cout << tr(X) << endl;                   // X(0,0) + X(1,1) + X(2,2)
cout << det(Y) << endl;
cout << "X: " << X.rows()                // X: 3 x 3
     << " x " << X.columns() << endl;
cout << X.identity() << endl;
cout << X << endl;
cout << X.transpose() << endl;
cout << X*Y << endl;
cout << X.vec() << endl;                 // vec operator
cout << X.kron(Y) << endl;               // Kronecker product
cout << X.dsum(Y) << endl;               // direct sum
cout << X.inverse() << endl;             // direct sum
cout << X.row(0) * Y.column(1) << endl;
cout << (X.column(0) | Y.column(0)) << endl; // scalar product
cout << (X.column(0) % Y.column(0)) << endl; // vector product
```