JManip.hh

The file `JManip.hh` is an include file with several auxiliary manipulators that can be used in conjunction with the STL `std::ostream` class. These are particularly useful for formatting numbers, floating point values and custom data structures. In the following, a few examples are provided.

For formatting numbers, the following code extracts will produce pairwise identical outputs.

```
const int i = 123456;

cout << setw(12) << left << i;
cout << LEFT(12) << i;

cout << setw(12) << right << i;
cout << RIGHT(12) << i;

cout << setw(12) << setfill('0') << i;
cout << FILL(12, '0') << i;
```

For formatting floating point values, the following code extracts will produce pairwise identical outputs.

```
const double x = 123.456;

cout << setw(12) << setprecision(5) << fixed << x;
cout << FIXED(12,5) << x;

cout << setw(12) << setprecision(2) << scientific << x;
cout << SCIENTIFIC(12,2) << x;
```

For lambda functions with capture, the wrapper `LAMBDA` can be used, e.g:

```
vector<int> V = { 1, 2, 3, 4};

cout << LAMBDA([v = V](ostream& out) { for (const auto& i : v) { out << " " << i; } }) << endl;
```

An internal print flag is also available that can be used to customize the formatting of user classes as follows.

```
struct C {

  friend inline std::ostream& operator<<(std::ostream& out, const C& object)
  {
    using namespace JPP;

    switch (getPrintOption(out)) {

    case SHORT_PRINT:
      return out << "C::shortprint";

    case MEDIUM_PRINT:
      return out << "C::mediumprint";

    case LONG_PRINT:
      return out << "C::longprint";
```

```
    default:
      return out << "C::undefined";
    }
  }
};
```

The print flag can be set with a designated manipulator. The following code extract:

```
  C c;

  cout << shortprint  << c << endl;
  cout << mediumprint << c << endl;
  cout << longprint   << c << endl;
```

will produce:

```
C::short_print
C::medium_print
C::long_print
```

To customize the format of data members in a data structure *a posteriori*, the template methods `getFormat` and `setFormat` can be used as follows.

```
struct A {

  A(const double value) : value(value) {}

  friend inline std::ostream& operator<<(std::ostream& out, const A& object)
  {
     const JFormat format(out, getFormat<A>(JFormat_t(5, 2, std::ios::fixed | std::ios::showpos)));

     return out << format << object.value;
  }

  double value;
};

struct B {

  B(const double value) : value(value) {}

  friend inline std::ostream& operator<<(std::ostream& out, const B& object)
  {
     const JFormat format(out, getFormat<B>(JFormat_t(12, 3, std::ios::scientific | std::ios::showpos)));

     return out << format << object.value;
  }

  double value;
};
```

In this, the argument at the template method `getFormat` corresponds to a default format. The data structure `JFormat_t` contains a number of format parameters. The list of parameters includes:

```
struct JFormat_t {
  int      width;
  int      precision;
  fmtflags flags;
  char     fill;
};
```

The data structure `JFormat` corresponds to a sentry object and derives from `JFormat_t`. The new format is set upon construction of this data structure and the previous format restored upon destruction.
The following code extract

```
    const double c = 12.34;

    const A a(c);
    const B b(c);

    cout << setprecision(3);

    cout << "A <" << a << ">" << endl;
    cout << "c <" << c << ">" << endl;
    cout << "B <" << b << ">" << endl;
    cout << "c <" << c << ">" << endl;

    setFormat<A>(JFormat_t(12, 3, std::ios::scientific));
    setFormat<B>(JFormat_t( 5, 2, std::ios::fixed));

    cout << "A <" << a << ">" << endl;
    cout << "c <" << c << ">" << endl;
    cout << "B <" << b << ">" << endl;
    cout << "c <" << c << ">" << endl;
```

will produce

```
A <+12.34>
c <12.340>
B <  +1.234e+01>
c <12.340>
A <   1.234e+01>
c <12.340>
B <12.34>
c <12.340>
```

As can be seen, the internal format of `cout << a` and `cout << b` can be modified whilst the format of `cout << c` is not affected.