

JLang

The directory JLang contains a variety of auxiliary classes. These reside in the corresponding name space JLANG. In the following, the template classes JType, JTypeList, JTYPELIST and JBool are briefly described. These constitute low-level classes for various evaluations at compile time.

The class JType is a simple place holder for a data type. A possible implementation of this class is:

```
template<class T>
struct JType {
    typedef T data_type;
};
```

It can be used for type dependent functionality without creation of an actual object. For example, the following method will return true for an int data type and else false.

```
template<class T>
bool get(JType<T>) { return false; }
bool get(JType<int>) { return true; }
```

This feature is used in e.g. method `getTreeParameters(JType<..>)` to define the configuration of the ROOT TTree corresponding to a given data type. In this way, the configuration of a set of classes can centrally be defined and subsequently be used in any application without requiring the implementations of these classes to be included, compiled and linked.

The concept of a place holder for a data type can be extended to list of data types. For this, the class JTypeList can be used. A possible implementation of this class is:

```
template<class JHead_t = JNullType, class JTail_t = JNullType>
struct JTypeList
{
    typedef JHead_t          head_type;
    typedef JTail_t          tail_type;
};
```

In this, the class JNullType is a dummy data structure that is used to signal the termination of a list. The main trick now is that one can recursively extend a list of data types.

For example:

```
typedef JTypeList<int, JTypeList<float, JTypeList<double> > > > typelist;
```

constitutes a list of three data types. There also is the class JRemove which can be used to remove a data type from a list. With the introduction of variadic template lists in c++11, a shorthand is available in the form of class JTYPELIST. For example:

```
typedef JTYPELIST<int, float, double>::typelist typelist;
```

The concept of type lists is used a.o. to define the data types subject to I/O in many applications.

The class JBool can be used to define a boolean value at compile time. A possible implementation of this class is:

```

template<bool __value__>
struct JBool
{
    static const bool value = __value__;
};

```

This class also offers some basic boolean algebra. As an example, the print capabilities of a class can be checked at compile time so that a compiler error is avoided and a message is printed at run time instead.

```

template<class T>
print(constT& object) {
    print(object, JBool<JStreamAvailable<T>::has_ostream>());
};

template<class T> print(constT& object, JBool<true>) { cout<< object << endl; }
template<class T> print(constT& object, JBool<false>) { cout<< "cannot print." << endl; }

```

In this, the class JStreamAvailable is used to check at compile time if the stream operators are defined for a given class. It has two data members, namely:

```

template<class T>
struct JStreamAvailable<T>
{
    static const bool has_istream;
    static const bool has_ostream;
};

```

These data members are set to true if the corresponding stream operation is allowed and else false. The auxiliary class STREAM can be used to conveniently capture this functionality in the actual stream operation.

```

A a;

cout << STREAM("?") << a << endl;

```

If the output stream operator is defined for class A, the value of a is printed; else a question mark.