# Template Classes

Templated classes are great for writing generic code for multiple types (e.g. the same elimination algorithm code for symbolic, discrete, and Gaussian elimination) without the drawbacks of virtual inheritance (which include rigid class interfaces, downcasting from returned base class pointers, and additional runtime overhead). Depending on how they're used, though, templates can result in very slow compile times, large binary files, and hard-to-use code. This section describes the “best practices” we have developed for gaining the benefits of templates without the drawbacks.

If you need to write generic code or classes, here are several programming patterns we have found to work very well:

## The “Templated Base, Specialized Derived” Pattern

This pattern is for when you have a generic class containing algorithm or data structure code that will be specialized to several types. The templated base class should never be used directly, instead only the specializations should be used. Some specialized types can be pre-compiled into the library, but the option remains to specialize new types in external libraries or projects.

### Base Class

We'll use FactorGraph as an example. It is templated on the factor type stored in it and has several specializations. The templated base class FactorGraph<class FACTOR> is divided into a header file (.h) and an “instantiation” file (-inst.h). The basic class structure is as follows.

// File FactorGraph.h

// Include a minimal set of headers. Do not include any '-inst.h' files (this is the key to fast compiles).

#include <boost/serialization/nvp.hpp>

...

**namespace** gtsam {

/\*\* Class description \*/

**template**<**class** FACTOR>

**class** FactorGraph

{

// Make 'private' any typedefs that must be redefined in derived classes. E.g. 'This' in the context of the derived class should refer to the derived class. These typedefs will be used only by the generic code in this base class.

**private**:

**typedef** FactorGraph<FACTOR> This; ///< Typedef for this class

**typedef** boost::shared\_ptr<This> shared\_ptr; ///< Shared pointer to this

// Make 'public' the typedefs that will be valid in the derived class.

**public**:

**typedef** FACTOR FactorType; ///< Factor type stored in this graph

**typedef** boost::shared\_ptr<FACTOR> sharedFactor; ///< Shared pointer to a factor

...

// Normally, data is 'protected' so the derived class can access it.

**protected**:

/\*\* Collection of factors \*/

std::**vector**<sharedFactor> factors\_;

// Make 'protected' all constructors, named constructors, or methods returning the base class type. These are not public - the derived class will call them and properly convert returned base classes to the derived class.

/// **@name** Standard Constructors

/// **@{**

/\*\* Default constructor \*/

FactorGraphUnordered() {}

/\*\* Named constructor from iterator over factors \*/

**template**<**typename** ITERATOR>

**static** This FromIterator(ITERATOR firstFactor, ITERATOR lastFactor);

/// **@}**

// Make 'public' standard methods that will be available in the derived class's API.

**public**:

/// **@name** Adding Factors

/// **@{**

/\*\* ... \*/

**void** reserve(**size\_t** size);

...

/// **@}**

};

}

### Derived Class