Storing Properties in Grouped Tagged Tuples

Roland Weiss and Volker Simonis

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{weissr,simonis}@informatik.uni-tuebingen.de

Abstract A technique is presented that allows one to store groups of properties in C++, and single properties out of these groups can later be accessed by their name. Our approach refines previous work in this area and is an example for the application of template metaprogramming [1]. Typical usage examples of the introduced class templates are internal representations of serialized data, well suited for semi-automatic as well as manual generation of the corresponding class types.

1 Introduction

Cartesian product types are a fundamental building block for composite types in virtually all modern programming languages. For example, they are present as record types in Pascal, Ada, and Modula-3, in C/C++ [8] as structs, and functional languages often support tuple types, e.g. ML [10] and Haskell [9].

However, explicitly defining a new class in C++ in order to create a simple record type without advanced functionality was perceived as an unnecessary heavyweight process. Numerous approaches for integrating lightweight tuples were developed. Most notably, Jakko Järvi introduced a tuple library [7] that allows access to its elements by index or type, which finally resulted in the Boost Tuple Library [5]. He identified handling of multiple return values as one of the main applications of his tuple library [6].

Emily Winch proposed a tuple variant with access to its elements by name [16]. This comes closer to a typical record type. The advantage of the technique presented by Winch is that parts of a class can be generated with template metaprogramming. Therefore, tedious and error-prone class operations like constructors and the assignment operator can be generated from a formal description. Furthermore, she describes an iterator interface for manipulation of the tuple's elements with algorithms that execute at compile time.

We will identify a shortcoming in her implementation that may lead to inconsistencies and propose an adequate solution, which relies on template metaprogramming and is completely transparent to the user. We then show how this basic record type can be extended in order to group common properties in a composite data structure that still allows flat, direct access to single properties. Frequently, such a data structure is useful when creating an internal representation of serialized data.

2 Named Objects Revisited

Winch shows in great detail how tuples of named objects can be created, manipulated and how they can help building complex classes on top of them. However, there is a fundamental problem in her approach for defining a tuple. This becomes apparent when looking at her introductory example:

⟨**Listing 1.** Code extracted from file `src/named_objects.cpp`, lines 9 to 12⟩ ≡

```
struct myBigClass {}; struct age {}; struct myDatabase {};
typedef makeVarlistType3<
  BigClass*, myBigClass, int, age, Database&, myDatabase
>::list VarlistType;
```

First, one has to define empty structs in order to introduce their names for accessing tuple elements. Then, these names are used in the tuple's type definition. There, one has to pair them with what we call an *implementation type*. The tuple will actually hold elements of the implementation type, and these elements can be easily referenced by their *name type* later on. The problem arises when one creates other tuples using the same name type. Usually, the pair of implementation and name types is fixed, otherwise the resulting data structures would become very confusing for the user. So one has to remember this type pairing to keep the related data structures consistent. This represents a typical maintenance nightmare.

How can we deal with this problem? The desirable situation would be to permanently tie an implementation type to its corresponding name type. As structs can contain type definitions, this is achieved without problems. More challenging is the creation of a tuple type consisting of elements whose types are determined by associated implementation types. Our solution is sketched in the next paragraphs.

The definition of a tuple functionally equivalent to the one presented in listing Listing1 now looks like this:

⟨**Listing 2.** Code extracted from file `src/named_objects.cpp`, lines 20 to 24⟩ ≡

```
struct myBigClass { typedef BigClass* type; };
struct age { typedef int type; };
struct myDatabase { typedef Database& type; };
typedef Tagged_Tuple<
  TypeList<myBigClass, age, myDatabase>::type > PropType;
```

We see how a name type is permanently associated with its implementation type by nesting it directly inside the name type's struct. When defining a tagged tuple[1], one simply has to list the name types. Notice that the name types are passed inside a type list. This saves us from explicitly denoting the number of names as in the constructor function `makeVarlistType3`. The type list constructor used is an extension of the Loki library [1] which is based on an idea contributed to Thomas Becker [2].

Type lists are central to our approach for solving the problem of computing the type of the implementation tuple. Loki provides a tuple type that features

[1] The term *tagged tuple* was coined by David Abrahams in this context.

a constructor accepting a type list which contains its elements' types. Unfortunately, we cannot use the type list passed in the `Tagged_Tuple` constructor, because therein the implementation types are wrapped inside their name types. We employ a special template metaprogram `ExtractTypes` that creates a new type list from a type list consisting of name types with nested implementation types. It simply walks over all elements in the instantiation type list `TL` and extracts each nested implementation type, which is appended to the new list recursively. With this template metafunction at hand, the implementation tuple's type is computed like this:

⟨**Listing 3.** Code extracted from file `src/Named_Props.hpp`, line 104⟩ ≡

```
typedef typename ExtractTypes<TL>::type types_tl;
```

This type list `types_tl` can now be used to instantiate the tuple `m_props` holding the actual elements. We face a final complication when implementing the tagged tuple's access methods. Access to an element is triggered by a name type given as instantiation parameter `PropT`, but the implementation tuple only knows its implementation types. We have to use the fact that an implementation type is located at the same position as its hosting name type in their corresponding type lists. Again, a template metaprogram computes the return type for us. The metafunction `return_t` expects three parameters: the name type for which the implementation should be located, and the type lists holding the implementation and name types, respectively. It returns the implementation type located at the same position as the name type. Now, we can define the mutating access function in terms of this helper function:

⟨**Listing 4.** Code extracted from file `src/Named_Props.hpp`, lines 137 to 141⟩ ≡

```
template <class PropT>
typename return_t<PropT, tuple_type, TL>::type at() {
  return Loki::Field<
    Loki::TL::IndexOf<props_tl, PropT>::value >(m_props);
}
```

At this point, we supply the same functionality as Winch's heterogenous list, but with a more consistent definition for the name and implementation pair types. Type lists and template metaprogramming were instrumental in making the required computations transparent for the user.

3 Groups of Tagged Tuples

We now move on to a data structure that is tailored towards a special kind of problem. When internal data in serialized, these data often consists to a large degree of so called properties, e.g. the JavaBeans specifications [3] lists properties as one of its three main constituents. Properties describe the section of an object's internal state that is made visible to other entities, this means they usually can be read and set with simple access methods. Compared to other members, they have simple dependencies and can be changed in isolation. The tagged tuple type described in the previous section is a good candidate for storing properties. In this section we present a convenient class template for combining groups of properties. This is especially useful if property groups are part of several components. Let us state the requirements we have on the class template `Named_Properties`.

1. Access to single properties is type safe, i.e. line numbers should be stored as integers, file names as strings, and so on.
2. Related properties can be grouped together, e.g. information relevant for debugging, or color information.
3. Groups of properties can themselves be combined to completely describe the visible part of a component's internal state exposed through properties.
4. Access to a single property should be possible simply by its name, i.e. the property group it belongs to should be deduced automatically.

The first two requirements are already fulfilled by `Tagged_Tuple`. Combining property groups is also achieved easily by putting tagged tuples into a tuple type themselves. The hardest part is providing *flat* access to single properties. This is not possible with standard record types, because one has to fully specify the target property, e.g. `Button.colors.foreground`, which selects a button's foreground color, which is part of the button's colors property group.

We will now develop the class template `Named_Properties`. It has one template parameter `TL`, which is a type list that should contain tagged tuple types. The named properties' single member is a tuple generated with this type list.

⟨**Listing 5.** Code extracted from file `src/Named_Props.hpp`, line 210⟩ ≡

```
template <class TL> class Named_Properties
```

In order to support the last requirement, we once again have to resort to extensive template metaprogramming. This is possible because we know at compile time which property groups make up the data structure, and we can look up the desired property inside these groups. The following listing shows how the element access method `at()` is realized.

⟨**Listing 6.** Code extracted from file `src/Named_Props.hpp`, lines 259 to 263⟩ ≡

```
template <class PropT>
typename return_t<PropT, tuple_type, TL>::type at() {
  return Loki::Field<IndexOfNP<TL, PropT>::value>(m_props).
         template at<PropT>();
}
```

The method's return type is computed with the local template metafunction `return_t`, which first determines the property group that contains the name type `PropT`, and then selects the implementation type at the name type's position. The same two-level process is applied in order to obtain the reference of the actual element. First, we select the property group containing the name type `PropT`, then retrieve the reference of the tuple element by this name. This can be seen directly in the body of method `at()` in listing Listing6. Figure 1 depicts this two-level process of determining the property's reference. The reference of the shaded element belonging to `data` will be bound to `x`.

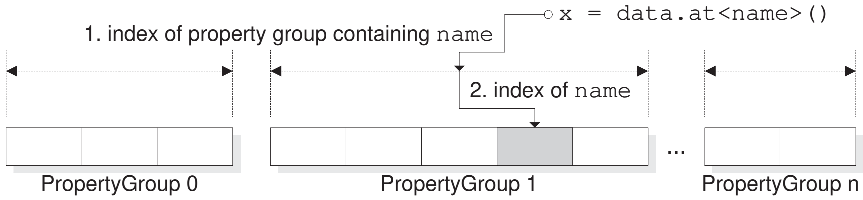

Figure1. The two-level selection process in method `at()`.

At the moment, named properties support the presented two level access to elements. There is no conceptual obstacle to extending them to three or more levels, but it seems questionable if such class templates make sense. Name conflicts are more likely to arise, and tracing the property groups is a major challenge for the user.

Finally, we have to consider the requirements on the instantiation parameters of the class templates and their method templates. The grouping approach relies on the fact that a name type is only part of one property group. If this condition is not met, accessing such a property will only locate its first appearance in the property groups given in the instantiating type list. Furthermore, if method `at()` is instantiated with a name type that is not present in any property group this will result in a compile time error, a reasonable reaction. The only problem with this behavior are the confusing error messages generated by some compilers.

We also want to mention an alternative approach for fulfilling the stated requirements of class `Named_Properties`. Grouping properties can be achieved by appending type lists that constitute groups of properties. Then, this type list can be used to instantiate a tagged tuple. It will contain all properties and supports flat access to them for free. This approach has its merits in its simplicity, because it does not need class `Named_Properties` at all. However, we loose the ability to use different meta operations on isolated groups of properties, as describe in Winch's paper [16].

4 Example Applications and Performance Tests

Named properties are part of the support library in a compiler project [15]. The project's central intermediate representation consists of an abstract syntax tree. Nodes of this tree data structure contain named properties, which combine a node's general and specific property groups. General properties fall into categories like debugging information, type modifiers, or cross references. Local properties are put into one group and complement the global groups which are reused in several node types. For example, the properties of an algorithm node are defined like this:

⟨**Listing 7.** Code extracted from file `src/Algorithm.hpp`, lines 50 to 53⟩ ≡

```
typedef Named_Properties< Loki::TypeList<
```

```
    id_property, idref_property, builtin_property,
    debug_property, access_mod_property>::type
  > algorithm_properties;
```

The named property is then aggregated by a generic syntax tree node, a technique similar to the one described in [14]. Named properties can be reused in several other application domains where the class types have a high percentage of property members. Examples are document management systems, or an internal representation of HTML nodes. In the HTML document type definition [11] *generic attributes* can be directly modeled with named properties. Of course, the same procedure can be applied to XML documents.

Finally, we want to compare the performance of our data structures generated with template metaprogramming to C++ classes written by hand. For this purpose, we use a little bench marking application. It first creates a huge vector, and then sets and reads all the properties of its element type, which is of course an instantiation of `Named_Properties`. Tables 1 and 2 summarize the results of the tests. The times in table 1 are given in microseconds and are computed by taking the arithmetic mean of five runs.

compiler	handwritten classes				named properties				AP
	create	write	read	sum	create	write	read	sum	
g++ 3.2	739	2089	2155	*4983*	1104	2750	2952	*6806*	1.37
g++ 3.2 -O2	613	1614	1672	*3899*	661	1710	1767	*4138*	1.06
Metrowerks 8.3	431	4068	877	*5376*	495	4192	991	*5678*	1.06
Metrowerks 8.3 -O4	276	3801	495	*4572*	340	3908	583	*4831*	1.06
Visual Studio 7.1	470	4068	877	*5376*	495	4192	991	*5678*	1.51
Visual Studio 7.1 -O2	470	994	461	*1925*	481	1015	473	*1969*	1.02

Table1. Benchmark results for comparing named properties to handwritten classes with a vector of 2 million elements, performed on a Pentium 4 machine (2GHz, 512 MB) running Windows XP.

Table 1 shows the results for three compilers, both with and without optimizations. We enumerate the results for initializing a vector, and subsequently writing to and reading from all its elements. Also, the sum of the runtimes is listed. In the last column the abstraction penalty (AP) is given, which is the ratio of dividing the runtime of the abstract version by the runtime of the one written at a lower level [12]. In our case, the code using handwritten classes represents the lower level version.

Table 2 lists the sizes of the vector's element data types, both for handwritten classes and those using named properties. This is done for all tested compilers. The sizes for debug and release versions did not differ.

The abstraction penalty for using the class template `Named_Properties` is very low for optimized builds, between 1.02 and 1.06. And even for code produced without optimizations an AP ranging from 1.06 to 1.51 is quite moderate. This

compiler	object size	
	handwritten classes	named properties
g++ 3.2	32	32
Metrowerks 8.3	56	56
Visual Studio 7.1	104	112

Table2. Object size for data types used in benchmarks (see table 1).

demonstrates that the tested compilers are effective at inlining the methods generated by the template mechanism.

The difference of the elements' object size is caused by the compilers' standard library implementation. The `std::string` class of the g++ compiler has size 4, Metrowerks' `string` class has size 12, and an object of class `string` in the Visual Studio amounts to 28 bytes. The named properties used in the benchmarks contain three strings, which accounts for the different object sizes. However, the object size of the handwritten classes and generated named properties are the same except for the Microsoft compiler[2], which is the primary observation to be made with respect to memory efficiency of named properties.

5 Conclusions

We have presented a technique that allows type-safe storage of groups of properties. Noteworthy for our implementation is the combination of class templates with metaprogramming on types. This integrated approach to code generation as provided by the C++ template mechanism has proven very successful for the development of efficient and type-safe components [4,13]. However, the actual incarnation of this metaprogramming environment leaves much to be desired, mainly motivated by the fact that it was a by-product not originally intended when designing C++'s template machinery. Further research should focus on identifying the typical needs for such a metaprogramming facility. The general technique could then be applied to languages other than C++.

References

1. Andrei Alexandrescu: *Modern C++ Design.* Addison-Wesley, 2001.
2. Thomas Becker: *STL & Generic Programming - Typelists.* C/C++ Users Journal, December 2002.
3. Graham Hamilton (Editor): *JavaBeans^{TM}, V1.01.* Sun Microsystems, 1997.
4. Scott Haney, and James Crotinger: *How Templates Enable High-Performance Scientific Computing in C++.* Journal of Computing in Science and Engineering, Vol. 1, No. 4, IEEE, July/August 1999.

[2] The C++ compiler packaged in Microsoft's Visual Studio 7.1 is their first C++ compiler that is able to handle advanced template metaprograms, therefore this peculiarity may disappear in subsequent releases.

5. Jaakko Järvi: *Tuple types and multiple return values.* C/C++ Users Journal, August 2001.
6. Jaakko Järvi: *Tuples and multiple return values in C++.* Turku Centre for Computer Science, Technical Report 249, March 1999.
7. Jaakko Järvi: *ML-style Tuple Assignment in Standard C++ – Extending the Multiple Return Value Formalism.* Turku Centre for Computer Science, Technical Report 267, April 1999.
8. JTC1/SC22 – Programming languages, their environment and system software interfaces: *Programming Languages – C++.* International Organization for Standardization, ISO/IEC 14882, 1998.
9. Simon Peyton Jones, and John Hughes (eds.): *Haskell 98: A Non-strict, Purely Functional Language.* Language Report, 1998. Available at `www.haskell.org`.
10. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen: *The Definition of Standard ML - Revised.* MIT Press, May 1997.
11. Dave Raggett, Arnaud Le Hors, and Ian Jacobs (editors): *HTML 4.01 Specification.* W3C Recommendation, December 1999. Available at `www.w3.org/TR`.
12. Arch D. Robertson: *The Abstraction Penalty for Small Objects in C++.* Workshop on Parallel Object-Oriented Methods and Applications (POOMA 96), Santa Fe, New Mexico, USA, February/March 1996.
13. Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine: *The Boost Graph Library. User Guide and Reference Manual.* Addison-Wesley Publishing Company, 2001.
14. Volker Simonis, Roland Weiss: *Heterogeneous, Nested STL Containers in C++.* LNCS No. 1755 (PSI '99): p. 263-267, Springer, 1999.
15. Roland Weiss: *Compiling and Distributing Generic Libraries with Heterogeneous Data and Code Representation.* PhD thesis, University of Tübingen, 2003.
16. Emily Winch: *Heterogenous Lists of Named Objects*, Second Workshop on C++ Template Programming, Tampa Bay, Florida, USA, October 2001.