

Heterogenous Lists of Named Objects

Emily Winch

Abstract

A heterogenous list of objects is introduced, in which the objects can be retrieved by name. An implementation of iterators for this list is discussed, along with algorithms that operate on them, and it is shown how these can be used to generate a substantial proportion of a class definition. It is demonstrated that the ability to attach meta-information to the objects in the list allows for code to be generated from higher level semantic specifications than has previously been possible.

1 Introduction

Heterogenous lists of types in C++ were introduced by Czarnecki and Eisenecker in [1]. These lists and the methods used to process them have more in common with those in Lisp and related languages than with the C++ `std::list`. Each list node has a head type and a tail type, where the head is the “data” and the tail can be another list type or an end marker type. This list of types is a compile-time construct, made up of nested typedefs. The implementation looks like this:

```
class Nil;
template<class T, class U = Nil>
struct List{
    typedef T Head;
    typedef U Tail;
};
```

A list containing an `int`, a `char` and a `bool` is written

```
typedef List<int, List<char, List<bool> > > myList;
```

The list is manipulated with a recursive template class, using a specialisation on `Nil` to terminate the recursion.

```

template<class List> struct Length;
template<> struct Length<Nil>{
    enum{ value = 0 };
}
template<class T, class U> struct Length<Typelist<T, U> >{
    enum{ value = 1 + Length<U>::value };
}

```

This has made possible a number of particularly interesting techniques. Several common design patterns vary in the types that they manipulate, and implementations of these patterns using lists of types are demonstrated in [2]. This was the first time such patterns had been encapsulated in a library rather than written out by hand each time they were required.

[3] showed an extension of this concept, where each `List<T, U>` holds an object of type `T` and type `U`, giving a “tuple” or heterogenous list of objects. The objects in the list can be accessed either by index or by type. However, this is of limited use. Indexed access does not lead to descriptive source code without the use of constants, which must be maintained separately from the list. Access by type can also be problematic, if the types in the list are not logically required to be unique.

This paper demonstrates a variation on the list of objects which allows objects to be accessed by name. The implementation is described in Section 3, and the following section demonstrates how functors, iterators and algorithms can be written to take advantage of these lists. Section 5 demonstrates the use of these techniques to implement copy constructors and assignment operators.

The exact operation to be performed on a variable is often determined by semantic information, for example, the “ownership” of the variable. Section 6 demonstrates how this semantic information can be attached to the variables and used to generate the correct code in each case.

2 Named objects

There are two particular situations in C++ where it is normal to have lists of named objects: the members of a class, and the parameters to a function. In both of these situations, it is common for the types of the objects not to be unique within the list.

Class members are nearly always accessed by name. The exception to this is brace initialisation, where only the index within the list is important. It is notable that C99 added “designated initialisers” to C, effectively allowing member access by name within brace initialisation as well [4].

There are many functions commonly forming part of class definitions that are normally written according to a simple set of rules, performing some operation on each member of the class in turn. These include `operator==`, `operator=`, `operator<<`, `operator>>`, destructors, and copy constructors. Storing the member variables of the class in a container makes it possible to use algorithms like those in the Standard Library to generate most of this code automatically. Members can then be added to the class without adding code to any of these functions.

Function parameters are also normally accessed by name. The ability to store the parameters in a

list of named objects allows a function to take a variable number of parameters in a flexible and typesafe manner. It can also simplify problems such as that of passing constructor parameters to a number of arbitrary mixin classes [5]. A mixin class could use names rather than order to retrieve its parameters from a list, passing the remainder on to a further mixin class.

3 Implementation

C++ does not have a mechanism for introspection that can discover the name of a variable in order to dispatch on it. However, templates provide a way to dispatch statically on type. Thus, it is possible to use types to represent names. The object list class is easily modified to take an extra type as a template parameter.

```
class Nil{};
template<typename T, typename N, typename U>
struct Varlist{
    typedef T Type;
    typedef N Name;
    typedef U Tail;
    Varlist(Type data, U u = Nil()) : data(data), u(u){}
    typename boost::add_reference<Type>::type get(){
        return data;
    }
    typename const_ref<Type>::type get() const{
        return data;
    }
    void set(typename const_ref<Type>::type data){
        this->data = data;
    }
    const U& tail() const{ return u; }
    U& tail() { return u; }

private:
    Type data;
    Tail tail;
};
```

Note the use of `boost::add_reference`, from the Boost library [6], which serves to add a reference if necessary, while not causing a reference-to-reference error if the type is already a reference. The `const_ref` template is similar, also adding a `const`-qualification if one is not present.

Since the syntax to declare an instantiation of `Varlist` is fairly unpleasant, helper classes are provided that take a number of template parameters and turn them into a `Varlist` type:

```

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

```

Similar functions are provided to build up the actual list object and return it for use.

```

VarlistType vars =
    makeVarlist<VarlistType>(pBigClass, 5, theDatabase);

```

4 Iterators, functors and algorithms

4.1 A Varlist iterator

Iterators for Varlists are similar to forward iterators for normal homogenous containers, but they are not interchangeable. It would not be possible to increment an iterator into a heterogenous container without somehow changing its type. Varlist iterators do not have an `operator++`: instead a `next()` function returns the next iterator. The iterators also contain typedefs for the variable name, the type pointed at, and the type of the next iterator, as returned by `next()`. These typedefs are contained in the iterator class, rather than in a traits class, since it is not possible for an iterator for a heterogenous container to be a built in type. Iterators for heterogenous containers are also comparable for equality *at compile time*, to facilitate the compile-time expansion of algorithms acting over those containers.

Two other kinds of iterators are also particularly useful with Varlists. A `FilterIterator` is constructed with normal `begin` and `end` iterators, but iterates through only those objects for which a user-supplied predicate is true. For example, the following code will zero only the pointers from `begin` to `end`:

```

// Assume that BeginIt and EndIt are the
// types of the two iterators, begin and end
MakeFilterIterator<BeginIt, EndIt, boost::is_pointer>
    MakePointerIterator;
typename MakePointerIterator::RET filter =
    MakePointerIterator(begin, end);
transform(filter, end, filter, zero());

```

A combining iterator is also required to implement the `operator==` of a class that uses Varlists. Dereferencing this iterator returns the result of applying a user-supplied binary functor to the objects pointed at by two other iterators. Following this paragraph is the code for the `operator==` of a

class containing a Varlist. Each ComboIterator is constructed with two iterators that point to equivalent positions in two containers, and an equality comparison functor. It is possible to accumulate the contents of the “container” produced by applying the functor to each pair of objects in turn. A `logical_and` functor is used with `accumulate`, producing the logical-and of the equalities of the objects, rather than the sum.

```
// assume standardEqual is a functor which
// compares two objects for equality; vars
// is the Varlist; ConstBeginIterator and
// ConstEndIterator are the types of the const
// begin and end iterators.

bool Person::operator==(const Person& other) const{
    ComboIterator<ConstBeginIterator, standardEqual>
        begin(vars.begin(), other.vars.begin(), standardEqual());
    ComboIterator<ConstEndIterator, standardEqual>
        end(vars.end(), other.vars.end(), standardEqual());
    return accumulate(begin, end, true, logical_and());
}
```

4.2 Functors

A functor describes an action which should be taken on each member of a list. The action will usually vary according to the type of the list member, for example, deleting pointers but leaving other types. For some functors, the action may vary according to the name of the variable. An integer variable representing a unique object identifier should not be copied in the same way as other integer variables.

It must therefore be possible to specialise functors for Varlists on the type of the variable, and sometimes the name. The functor class itself can not take template parameters, since it should be possible for all functors to carry information (such as the return type) in the form of typedefs that should be accessible before the template parameters are known. It is the `operator()` of the functor that takes the template parameters. However, it is not possible for the user to overload this member function in order to provide specialised behaviour. The member function will therefore usually forward to a further internal functor where the class itself can take template parameters.

The `operator()` of functors for Varlists takes the variable type as a template parameter, and may also take the name. Often a functor taking the name will forward directly to a functor specialised by the type, leaving the user to specialise either the name-dispatched functor or the type-dispatched functor in order to customise the behaviour.

4.3 Algorithms

Varlist algorithms, while very different in implementation, are identical in call syntax to those in the Standard Library. Where the standard algorithms compile into a loop which iterates through

objects at runtime, these produce an unrolled version of the loop, by recursive inlining of functions. A destructor produced by using a compile-time `for_each` and a deleting functor should produce the same object code as that of a destructor with the same functionality written out by hand.

For example, the implementation of `for_each` looks like this:

```

struct do_nothing{
    template<class T, class U, class V>
    V operator()(T&, U&, V& v){
        return v;
    }
};

template<typename BeginIt, typename EndIt, class Op>
inline Op for_each(BeginIt begin, EndIt end, Op op){
    typedef typename BeginIt::template equal<EndIt> tester;
    // if begin is equal to end, do nothing, otherwise, call
    // do_for_each
    typedef typename IF<tester::value,
                        do_nothing,
                        do_for_each
                        >::RET thingToDo;
    return thingToDo()(begin, end, op);
}

struct do_for_each{
    template<class BeginIt, class EndIt, class Op>
    inline Op operator()(BeginIt begin, EndIt end, Op op){
        // Do the business
        op.template operator(<
            typename BeginIt::PointeeType,
            typename BeginIt::VariableName
            >(*begin,
            typename BeginIt::VariableName());
        // Recurse to process the rest of the list
        return for_each(begin.next(), end, op);
    }
};

```

While this code covers a large number of lines compared to the standard `for_each`, its operation is very simple. If the two iterators are the same, it returns having done nothing. Otherwise, it performs the operation on the dereferenced first iterator, and calls itself on the remainder of the list. The termination condition for the recursion is evaluated in a typedef at compile time, to allow the compiler to recursively inline the code.

Notice that `do_for_each::operator()` explicitly specifies the template parameters to the `operator()` of the functor. This enables the writer of the functor to use `boost::add_reference` to generate the parameter type, which would prevent the compiler from being able to deduce the type from the arguments.

5 Using a Varlist to help generate copy constructors and assignment operators

A non-trivial class will usually have an assignment operator, which is written out by hand, assigning each member variable one by one. A simple addition of a member variable to the class now requires further changes to be made, allowing bugs to be introduced.

Assignment operators must be written by hand because the compiler can not determine how to do it. For any given pointer, should we copy the object pointed to, or merely the pointer? Nevertheless, there are simple rules that normally apply. Pointers are usually owned, and should be deep copied, whereas references are not. Sometimes pointers are divided into two sets: those that are owned, and those that are not. This is meta-information, which with a Varlist can easily be attached to the variables themselves. Section 6 goes into more detail on how this may be implemented and used.

Different classes may use different rules, but it is usually the case that within a class the behaviour is consistent. Where it is not, there are two possible reasons. Some types of object cannot be copied normally, for example, where the copy constructor is private but a Clone function is provided. In addition, there are variables where a normal copy may be possible but is semantically incorrect. This kind of customisable behaviour is easily achieved with a Varlist-style functor.

There are also a number of issues related to exception safety that surround assignment operators. Strongly exception-safe assignment operators are, in more complex classes, impossible to achieve without containing all the members in a “Pimpl” [7], such that one set of members can be swapped with another in an operation that will not throw [8]. This, however, incurs the overhead of indirection, and of allocating all the members on the heap. Varlists support both options. The Varlist class takes a *copy policy* template parameter, allowing the user to choose whether the Varlist should own its contents, copying and deleting it using functions from the policy class, or whether the user gains control over the contents, allowing `auto_ptr`s to be used in the copying process to gain strong exception safety. Even where the user does take responsibility, the strongly exception-safe copying functionality can be encapsulated in a library function that turns a shallow copy of a list into a deep copy.

```

// assume standardDelete to be a functor that
// deletes only pointers
// and VarlistType is the type of vars, the list of
// variables.
Person& Person::operator=(const Person& other){
    Person temp(other);
    std::swap(vars, temp.vars);
    for_each(vars->begin(), vars->end(), standardDelete());
    return *this;
}

Person::Person(const Person& other){
    std::auto_ptr<VarlistType>
        pVars(new VarlistType(*other.vars));
    // turn the shallow copy into a deep copy
    deepCopy(other.vars->begin(), other.vars->end(),
            pVars->begin(), pVars->end());
    vars = pVars.release();
}

```

6 Variable meta-information

Since this technique implements variable names using classes, it is easy to add meta-information to those variables in the form of further typedefs or enums. It is common for variables to have conceptual information attached to them, particularly regarding ownership and access. Normally that information is implicit. Only some of the variables are deleted in the destructor, or only some variables have accessor or modifier functions. It is the responsibility of the programmer to make sure that the code matches the conceptual information.

With the Varlist approach, the meta-information can be added explicitly to each variable. This allows the functors used on the variables to alter their behaviour in the light of that information. For example, if all the pointers in a class carry information about whether they are owned by the class, the functors used for copying and deletion can deep copy or delete only the owned pointers. A class whose variables contain an enum indicating whether they are readable and writable can have one accessor function and one modifier function, taking the variable name as a parameter, and these functions can assert at compile time that the “permissions” are correct. This is a simple way of implementing the concept of “properties”. It retains encapsulation by enabling property access only through functions, while allowing those functions to be specialised for individual properties, for example, if a property ceases to be directly represented by a variable within the class.

Generally, the Varlist technique allows class functionality to be automatically generated from a specification of semantic information that normally remains implicit.

7 Where Varlists can be helpful

Varlists are suitable for classes with a large number of member variables and several functions which require iteration over the member variables: particularly where the number or type of member variables may change over time. The motivating example for this work was a set of classes containing different kinds of internet settings information. Each class contained a large group of related data, and was able to internalise and externalise it in two different ways. Many of the variables needed to be deleted in the destructors, and an `operator==` was used for testing the internalisation and externalisation. Adding one new member variable required the modification of all these functions, and forgetting to change the `operator==` could result in the test code failing to identify any problems with the changes. The code could have been substantially smaller and more maintainable had the technique presented here been used.

Varlists are also suitable for classes with many member variables that do not necessarily have functions which iterate over the variables, but that have a public interface consisting of a large number of non-virtual get and/or set functions. The Varlist offers an ideal opportunity to consolidate this interface into one get function and one set function, taking the variable name as a template parameter. Compile time asserts can be used along with meta-information in the variable to enforce read/write “permissions” on the variables in question. Public/protected/private access is maintained by the access specified for the type representing the variable name.

8 Conclusion

The relatively simple step of adding name-based access to a heterogenous list of objects greatly enlarges the possibilities for its use. Several interesting techniques are possible, particularly in the area of class definition generation. In all cases, the complex code can be supplied to users in a library that exposes a simple API similar to that of Standard Library containers and algorithms.

References

- [1] Krzysztof Czarnecki and Ulrich Eisenecker. *Metalisp*.
<http://home.t-online.de/home/Ulrich.Eisenecker/meta.htm>.
- [2] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.
- [3] Jaakko Järvi. *Tuples and Multiple Return Values in C++*. Technical Report 249, TUCS, March 1999.
- [4] ISO/IEC. *C Standard*. 1999.
- [5] Ulrich W. Eisenecker, Frank Blinn, and Krzysztof Czarnecki. *A Solution to the Constructor-Problem of Mixin-Based Programming in C++*. In First Workshop on C++ Template Programming, Erfurt, Germany, October 10, 2000.
- [6] *Boost libraries*. <http://www.boost.org>.
- [7] Herb Sutter. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000.
- [8] Herb Sutter. *Guru of the Week 59*.
<http://www.gotw.ca/gotw/059.htm>.