Generalized Arithmetic in C++*

Timothy A. Budd
Department of Computer Science
Oregon State University
Corvallis, Oregon
97331
budd@cs.orst.edu

July 10, 1998

Abstract

A generalized arithmetic package allows a programmer to think of numbers as abstract quantities, without regard to whether they are represented as integers, floating point values, fractions, complex numbers, or even polynomials. In this paper we describe two implementation techniques that can be used to produce a generalized arithmetic package in the programming language C++, and evaluate the advantages and disadvantages of the two approaches. The second approach uses a technique called multiple polymorphism that has applicability to many other problem areas.

Keywords: C++, Object Oriented Programming, Arithmetic, Polymorphism

1 Introduction

One of the philosophical principles of object oriented programming is that objects should be characterized by their behavior rather than by their structure. Distinct types of objects that share a common behavior, in C++ terms that implement a set of virtual methods derived from a common parent class, should be logically interchangeable. In everyday discourse one of the more common and interesting situations where this occurs involves the use and manipulation of numbers and quantities. We use a common set of operations, such as addition and subtraction, on a variety of different forms of values, such as integers, rational numbers, floating point approximations, sometimes even complex numbers and polynomials. In large measure we do this without concern for the particular form the quantity takes, even mixing different forms with impunity. A mathematician would not balk, for example, at adding an integer to a polynomial containing coefficients represented as complex numbers.

It is an interesting experiment, therefore, to see if one can write a set of C++ classes that come close to corresponding to this everyday usage of numbers. In the next section we outline two different

*To appear in *Journal of Object-Oriented Programming*

techniques for producing such values. The majority of this paper is thereafter devoted to comparing and contrasting the implementation techniques used in these systems. The techniques described, Coercive Generality and Double Polymorphism, are not limited to use only in arithmetic. Similar situations occur whenever dynamic lookup must be based on two or more arguments, rather than only a single receiver. Ingalls [Ing86], for example, describes the display of multiple graphics objects on different display ports.

2 Generalized Arithmetic

The abstract specifications for our generalized arithmetic package are easy to state. We want to define a class of objects, called Number, that can represent abstract quantities. At the very least these should be able to represent the concrete values directly manipulated by the underlying C language and hardware, that is integers and floating point values. Thus a declaration, such as the following, should create a quantity n representing an integer value 3, and a quantity f representing the floating point value 3.1415926.

```
Number n = 3;
Number f = 3.1415926;
```

We can perform operations on numbers to generate new numbers. These may have the effect of altering the underlying type of the number. For example, computing n / 3 should produce a fractional number. The expression f / 3, on the other hand, should produce a new floating point number. Since C++ applies the conventional rules of precedence to arithmetic operators, if i is a complex number representing the value with zero real part and one imaginary part, then 3 + 4 * i would produce a complex number representing 3+4i.

For the purposes of this experiment we decided to implement only the four arithmetic operations addition, subtraction, multiplication and division, in the context of the three classes Integer, Fraction, and Float. In section 3.2 we will discuss the difficulty of adding new operations or new classes of numbers.

Storage allocation for automatic variables in C++ is performed as part of the declaration process. Since different types of numbers occupy different amounts of storage, it is not possible for the single class Number to maintain storage for all the possibilities. Instead, instances of Number will simply maintain a pointer to an instance of another class, called InternalNumber. A portion of the class declaration for Number is shown in Figure 1 (the complete code for both versions are given in the appendix). The class definition provides rules for converting C constants of type integer and double into instances of Number, a destructor to recover the storage used by the InternalNumber when numbers are deleted, and friend functions to perform the arithmetic operations. The classes Integer, Fraction and Float are all implemented as subclasses of the class InternalNumber. The friend functions implementing the arithmetic operations merely throw them back to the instances of Internal Number. These functions are shown in Figure 2.

As might be expected, the most difficult aspect of implementing the generalized arithmetic package is handling operations where the arguments are of dissimilar types. The next three subsections will describe two entirely different approaches to this problem, and explore one technique which

```
class Number {
        InternalNumber *val;
        ~Number() { delete(val); }

    public:
    /* construction */
        Number() { val = new InternalNumber();}
        Number(int i) { val = new Integer(i);}
        Number(double d) { val = new Float(d);}
        Number(InternalNumber *v) { val = v;}

    ...
    /* arithmetic */
        friend Number operator+(Number& x, Number& y);
        friend Number operator-(Number& x, Number& y);
        friend Number operator*(Number& x, Number& y);
        friend Number operator/(Number& x, Number& y);

    /* printing */
        void streamon(ostream& s)
        { (*val).streamon(s); }
    };
```

Figure 1: Declaration for Class Number

```
Number operator+(Number&x, Number& y)
                { return *(new Number (*(x.val) + *(y.val))); }
Number operator-(Number&x, Number& y)
                { return *(new Number (*(x.val) - *(y.val))); }
Number operator*(Number&x, Number& y)
                { return *(new Number (*(x.val) * *(y.val))); }
Number operator/(Number&x, Number& y)
                { return *(new Number (*(x.val) / *(y.val))); }
ostream& operator<<(ostream&s, Number& n)
                { n.streamon(s); return s; }
```

Figure 2: Friend Functions for Numbers

```
class Float : public InternalNumber {
            double val;
            int generality() { return 30; }
    public:
            ...
    InternalNumber *operator+(InternalNumber& y)
            { if (y.generality() == 30)
                    {Float *f = (Float *) &y;
                            return new Float(val + f→val);}
            else {
                    return InternalNumber::operator+(y);
            }}
    ...
    InternalNumber *coerce(InternalNumber& y)
            { return y.asFloat(); }
    ...
}
```

Figure 3: Addition in Class Float

might at first appear to offer a promising solution but which does not. This will then be followed by a comparison of the advantages and disadvantages of the two successful techniques.

2.1 Coercive Generality

Coercive generality is the technique used by the Smalltalk-80 system described in the "blue book" [GoR84], and in the Little Smalltalk system [Bud87]. In this scheme each type of number is assigned a "generality index". Two numbers are of the same class if they have the same generality index. If not, the number with the lower generality index is modified (coerced) into being compatible with the number with the higher generality index. In our system we used the values 10, 20 and 30 for the generality index values for Integer, Fraction and Float, respectively. The generality value can be obtained by invoking the function "generality" on an instance of InternalNumber.

Instances of each of the derived classes test the generality of an argument, performing the operation if it matches the generality of the first argument. If not, the message is passed up to the method in the superclass. The code for addition in the class Float, shown in Figure 3, illustrates this behavior.

Control reaches the method in class InternalNumber only if two arguments are of different types. The generality values are consulted for each of the two arguments to determine which is "more general". The more general value is then passed the message "coerce", along with the other value as the argument. The coerce routine must alter the argument so that it is compatible with the receiver. Once the coerced result is returned, the original operation is then retried. Presumably this time the appropriate value is generated. The code to perform this is shown in Figure 4.

```
class InternalNumber {
    ...
    virtual InternalNumber *operator+(InternalNumber& y)
        { if (generality() < y.generality())
            return (*(y.coerce(*this)) + y);
        else return *this + *coerce(y); }
    ...
}
```

Figure 4: Addition code in Class InternalNumber

Each class must provide a means for coercing objects of lower generality into one compatible with the receivers class. In some cases, such as class Fraction, this involves merely creating a new object (a Fraction with a denominator of one). In other cases, such as class Float (Figure 3), this involves sending yet another message, asFloat, to the object to be converted. Classes of lower generality than Float must provide some means of responding to this message.

It is important to note that the technique of Coercive Generality requires the various types of numbers to form a linear hierarchy. In simple cases this is easy; for example it is clear that floating point values are more general than integers. More complicated cases are less clear cut; for example what is the relationship between complex numbers and polynomials? The technique of Double Polymorphism, described in the next section, avoids the necessity of forming this linear ordering.

2.2 Double Polymorphism

In a 1986 paper at the first OOPSLA (object oriented programming systems, languages and applications) conference, Daniel Ingalls presented an interesting technique for handling polymorphism in several variables [Ing86]. In simple terms, we make use of the automatic polymorphism facility implicit in message passing by making each argument, in turn, a receiver for a message and encoding the types of the remainder of the arguments in the message selector. Thus when a floating point value receives the message "+", it reverses the arguments, passing to the argument that it received the message "addToFloat" (Figure 5). The type of the argument is now explicit in the message selector. Each alternative class must then provide a mechanism whereby it can be added to each possible type of argument.

Using this scheme requires no generality numbers. It is not necessary for types of numbers to form a linear hierarchy. The cost, however, is that each type of number must have explicit information about each other type of number. One could argue that it is bad object oriented programming style to have knowledge about each number class distributed over all other classes, however it is largely the case that this knowledge was already present implicitly in the "coerce" messages of the generality technique.

```
class InternalNumber {
        ...
        virtual InternalNumber *operator+(InternalNumber& y)
                {return this;}
        virtual InternalNumber *addToFloat(InternalNumber& y)
                {cout << "subclass should provide\n"; return this;}
        };

class Float : public InternalNumber {
        double val;
    public:
        Float(double d) { val = d; }

        ...
        InternalNumber *operator+(InternalNumber& y)
                { return y.addToFloat(*this);}
        InternalNumber *addToFloat(InternalNumber& y)
                {Float *f = (Float *) &y;
                return new Float(val + f→val);}
        };

class Integer : public InternalNumber {
        int val;

        InternalNumber *operator+(InternalNumber& y)
                { return y.addToInteger(*this);}
        InternalNumber *addToFloat(InternalNumber& y)
                { return (*asFloat()).addToFloat(y);}
        };
```

Figure 5: Addition Using Double Polymorphism

```
class Integer;
class Number {
      public:
                virtual int operator+(Number& x)
                        { cout << "in Number,Number +"; return 7;}
                virtual int operator+(Integer& x)
                        { cout << "in Number,Integer +"; return 7;}
                };
class Integer : public Number {
      public:
                int operator+(Number& x)
                        { cout << "in Integer,Number +"; return 7;}
                int operator+(Integer& x)
                        { cout << "in Integer,Integer +"; return 7;}
                };
main() {
      Number *a;
      a = new Integer();
      (*a) + (*a);
      }
```

Figure 6: Why Overriding Is Not Sufficient for Mixed Mode Arithmetic

2.3 Overloading

The language C++ allows programmers to overload the arithmetic operators (a fact we have already used in both of the solutions presented). It is instructive to note why we cannot simply use this mechanism to achieve the effect we desire. The reason is that, while the message passing paradigm insures that a choice of method is found based on the class of the receiver, the selection of a method to be used in an overridden operator is based on the declared (static) type of the arguments and not the current (dynamic) type, which may be a derived subclass. This is illustrated by the simple example program shown in Figure 6, which is a very simplified version of our Number hierarchy. If a variable a is declared to be a pointer to Number, but nevertheless contains a pointer to an instance of a derived class Integer, the selection of a method to be executed in the expression (*a) + (*a) will be first based on the dynamic (run time) type of a for the first (receiver) argument and second on the static (declared) type of a for the argument. This the resulting value printed will indicate the method selected is "Integer,Number" and not "Integer,Integer".

Thus even if we directly defined methods for addition, for example, with all possible pairs of types of arguments, only the method where the argument is declared the general class "InternalNumber" would ever be executed.

While overloading cannot be used to directly solve the problem of selecting the right method from the class hierarchy, it can be used to reduce the number of names required to implement

	Coercive Generality		*Double Polymorphism*	
	methods	lines of code	methods	lines of code
Class InternalNumber	9	39	19	50
Class Float	6	49	18	52
Class Fraction	9	63	16	51
Class Integer	8	49	19	55
Total	32	200	72	208

Table 1: Lines of Code, Number of Procedures

double polymorphism. For example, we can replace the methods addToInteger(InternalNumber& x) and addToFloat(InternalNumbea& x) with addTo(Integer& x) and addTo(Float& x). However this reduction is actually an illusion; both from the compilers point of view and from the perspective of the number of lines of code that must be written the two are the same. Interestingly, when we experimented with a solution implemented in this fashion it was consistently about 10% slower than the version encoding the types of arguments in the method names.

3 Comparisons

Both the solutions described in previous sections implement the same functionality. Thus a comparison of the two techniques cannot be based on any differences in input/output behavior. Instead, we will examine a number of secondary issues, such as size, execution speed, and the ease with with changes and extensions can be implemented.

3.1 Size and Speed

Size can be measured in a variety of ways. From the programmers point of view, the most direct measurement of size involves the number of source lines, or possibly the number of procedures (methods) necessary to implement the system. These are shown in Table 1, which also breaks these figures down into quantities associated with each class. As can be seen from these statistics, the double polymorphism approach requires many more procedures; in this case over twice as many methods are defined using that approach. On the other hand, the procedures are generally very short, so that the final number of source lines is approximately the same for the two techniques.

Given the extensive use of in-line functions in C++, statistics such as lines of code or number of procedures are less meaningful than they might be for some other languages. A function that is expanded in-line will not impose the overhead of a procedure call, and in most cases the improvement in understandability is worth the slight increase in code size. Other characteristics of the program, such as object program size and run time performance, are less influenced by static program layout. Figures for object size and executing timings are shown in Table 2. For the purposes of timings, we used the following simple main program, gathering statistics by averaging repeated runs of the program on a Microvax II. As one might expect from the smaller number of procedures, the code for the generality version is smaller, about three forths the size of the double polymorphism

	Coercive Generality	*Double Polymorphism*
object size (bytes)	7828	10012
execution time (seconds)	11.9	9.8

Table 2: Object Size and Execution Time

version. Despite this fact, the double polymorphism version is consistently about 25% faster than the generality version.

```
main() {
        Number n = 3;
        Number m = 5.5;
        Number p;
        int i;

        for (i = 1 ; i < 10000; i++)
            p = (5 / n) + m ;
        }
```

One explanation for the superior performance of double polymorphism may be the streamlined manner in which type conversion is handled. Consider adding an integer to a floating point value. Using generality, the addition method in class Integer will first be executed. This invokes the generality function, before passing the addition message up to the class InternalNumber. The method in InternalNumber will invoke generality twice more, then call coerce. The method for coerce in class Float will call asFloat, before finally trying the addition method and producing the answer. Thus eight procedure calls are necessary to produce the proper value. Using double polymorphism, on the other hand, invoking the addition routine involves sending the message addToInteger, which will in turn in class Float invoke the method asFloat and addToFloat, for a total of only four procedure invocations.

Even in the situation where the arguments agree in form one can argue that the execution time performance of double polymorphism is no worse than that of generality. In generality, one additional procedure invocation will be necessary to establish the generality number of the argument. Using double polymorphism, one additional instruction is necessary to reverse the arguments. Given this, one would expect the execution time behaviors of the two systems to be closer if conversions are not required. This is borne out by empirical observations, where a program which merely added together two integers 10,000 times required 2.4 seconds using generality and 2.3 seconds using double polymorphism.

3.2 Extensibility

The three types of numbers we have described are only a small part of the set of numeric objects one might consider implementing. Other possibilities include infinite precision integers, complex numbers, polynomials, or even special types of numerical forms such as degrees or radians. Similarly there are many other operations, such as relational comparisons or logical operators, that one might

	Coercive Generality	*Double Polymorphism*
Class InternalNumber	0	OP
Class Float	1	OP
Class Fraction	1	OP
Class Integer	1	OP
New Class	2 + OP	OP × CL
TOTAL (approximately)	2 + CL + OP	2 × OP × CL

OP = number of operators
CL = number of classes

Table 3: Methods Required if a new Class is Added

wish to add to the system. Thus another way one might compare these two approaches is the ease with which they can be extended to include new classes or new operators.

3.2.1 Adding New Classes

When generality is used, adding new classes is relatively easy. It is not necessary to change the base class InternalNumber at all. In the new class, methods for producing the generality index and for coerce must be provided, as well as methods for each of the supported operations. A technique must be provided to coerce each value of lower generality into the new type, and to convert the new type into values of higher generality. How this is accomplished will vary from situation to situation, but usually it can be accomplished by adding about one new method to each existing class.

Adding a new class when double polymorphism is used is much harder. The base class InternalNumber defined methods for each pair of class and operator. These must be redefined in each derived subclass. Thus if a new class is added, *OP* methods, where OP is the number of operators being supported, must be defined in each existing class. In addition, all OP × CL pairs must be defined in the new class. Thus while for generality the number of new methods that need to be defined is approximately the sum of the number of classes and the number of operators, for double polymorphism this figure is the product of these two quantities (Table 3).

3.2.2 Adding New Operations

The simplicity of the generality approach in comparison to double polymorphism is still apparent even if one considers only the task of adding a new operator to an existing collection of classes. Using generality, it is only necessary to add the method for the operator in each existing class (assuming the already provided techniques for coercion are sufficient). The the total number of new methods is roughly proportional to the number of classes. Using double polymorphism, on the other hand, a method must be provided in each class to handle the operator and an argument from each of the other classes. Thus the number of new methods is proportional to the square of the number of classes (Table 4).

	Coercive Generality	*Double Polymorphism*
Class InternalNumber	1	CL + 1
Class Float	1	CL + 1
Class Fraction	1	CL + 1
Class Integer	1	CL + 1
TOTAL	CL	CL × CL

CL = number of classes

Table 4: Methods Required if a new Operator is Added

4 Conclusions

In many problems a situation can occur in which there is variation in two or more parameters, for example an event handler may have both multiple event types and multiple event handlers, or a graphics system may have multiple display objects and different display ports, or a numeric package can permit both the left and right arguments to arithmetic operations to vary among several types. From a programming point of view, these can be handled by dynamic lookup based on two or more objects. Most object oriented languages, such as C++ or Smalltalk, support dynamic lookup based only on a single object, the receiver[1].

We have described two approaches to dealing with the problem of dynamic lookup based on two arguments in the context of the problem of generalized numbers. The technique of coercive generality might be considered the more "traditional" approach, or at least to possess the claim of earlier development. The generality approach is simpler, in terms of lines of code, number of methods, object size, and ease of extension. Nevertheless, the newer approach of double polymorphism is superior in one very important respect; it is faster in execution. Given that the number of operators available for overridding is fixed by the language, and that the number of classes is likely to be in practice relatively small, this last statistic may well be the only important measure in most programmers' minds.

Operations using generalized numbers can never hope to be as efficient as operations using the native concrete types (integers and doubles) provided by the underlying hardware and C system. Thus one would probably not used a generalized number as, for example, a loop counter. Nevertheless, there are situations where the more general facility is desirable.

The utility of the double (or even multiple) polymorphism technique is not limited to its use with mixed mode arithmetic. As Ingalls describes in his original paper [Ing86], it can be used any time you have arguments that can vary in two or more positions for a method, and the type of the arguments cannot be determined at compile time. Such situations can easily occur, for example, if you have multiple types of picture objects that can be displayed on multiple types of output devices. Thus the technique is one that should be more widely known and used by programmers working in object oriented languages.

[1] To the authors knowledge, only the object oriented language CLOS directly supports dynamic lookup based on two or more objects [Kee89].

Acknowledgements

A number of people provided useful comments on an earlier draft of this paper, these include Frank Griswold, Rajeev Pandey, Don Pardo and Bjarne Stroustrup. Jim Adcock pointed out the use of overloaded arguments instead of using multiple method names described in section 2.3.

References

[Bud87] Budd, Timothy A., "A Little Smalltalk", Addison-Wesley, 1987.

[GoR84] Goldberg, Adele and Robson, David, *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1984.

[Ing86] Ingalls, Daniel H.H., A Simple Technique for Handling Multiple Polymorphism, *Sigplan Notices*, Vol 21(11): 347-349, November 1986.

[Kee89] Keene, S. E., *Object-Oriented Programming in Common Lisp*, Addison-Wesley, 1989.

Appendix 1: The Coercive Generality Code

While the following code shows the viability of the ideas, it is still somewhat incomplete. Memory management could be improved, for example. Similarly, fractions should be divided by their greatest common divisor, and fractions with a denominator of 1 should be made into integers. Finally a realistic system would implement many more operations (such as relational and logical operators).

```
/*
        generalized arithmetic using coercive generality
        written by Tim Budd
        Oregon State University
        Corvallis, OR 97331
/

# include <stream.h>

class InternalNumber {
        protected:
                virtual int generality() { return 0; }

        public:

        /* printing */
                virtual void streamon(ostream& s) { s << " undefined "; }
        /* conversion */
                virtual InternalNumber *coerce(InternalNumber& y)
                        { return &y;}
                virtual InternalNumber *asFloat() ;
                virtual InternalNumber *asFraction() ;
        /* addition */
                virtual InternalNumber *operator+(InternalNumber& y)
                        { if (generality() < y.generality())
                                return (*(y.coerce(*this)) + y);
                         else return *this + *coerce(y); }
        /* division */
                virtual InternalNumber *operator/(InternalNumber& y)
                        { if (generality() < y.generality())
                                return (*(y.coerce(*this)) / y);
                         else return *this / *coerce(y); }
        /* multiplication */
                virtual InternalNumber *operator*(InternalNumber& y)
                        { if (generality() < y.generality())
                                return (*(y.coerce(*this)) * y);
                         else return *this * *coerce(y); }
        /* subtraction */
                virtual InternalNumber *operator-(InternalNumber& y)
                        { if (generality() < y.generality())
                                return (*(y.coerce(*this)) - y);
                         else return *this - *coerce(y); }
                };

class Float : public InternalNumber {
                double val;
                int generality() { return 30;}
        public:
                Float(double d) { val = d; }

        /* printing */
                void streamon(ostream& s) { s << " Float " << val; }
        /* conversion */
                InternalNumber *coerce(InternalNumber& y)
                        { return y.asFloat(); }
        /* addition */
```

```
                InternalNumber *operator+(InternalNumber& y)
                        { if (y.generality() == 30)
                                {Float *f = (Float *) &y;
                                  return new Float(val + f→val);}
                         else {
                                 return InternalNumber::operator+(y);
                                }}
        /* division */
                InternalNumber *operator/(InternalNumber& y)
                        { if (y.generality() == 30)
                                {Float *f = (Float *) &y;
                                  return new Float(val / f→val);}
                         else {
                                 return InternalNumber::operator/(y);
                                }}
        /* multiplication */
                InternalNumber *operator*(InternalNumber& y)
                        { if (y.generality() == 30)
                                {Float *f = (Float *) &y;
                                  return new Float(val * f→val);}
                         else {
                                 return InternalNumber::operator*(y);
                                }}
        /* subtraction */
                InternalNumber *operator-(InternalNumber& y)
                        { if (y.generality() == 30)
                                {Float *f = (Float *) &y;
                                  return new Float(val - f→val);}
                         else {
                                 return InternalNumber::operator-(y);
                                }}
                };

class Fraction : public InternalNumber {
                InternalNumber *top, *bottom;
                int generality() { return 20;}
        public:
                Fraction(InternalNumber *t, InternalNumber *b)
                        { top = t; bottom = b; }

        /* fraction specific */
                InternalNumber *reciprocal()
                        { return new Fraction(bottom, top); }

        /* conversion */
                InternalNumber *asFloat()
                        { return (*(*top).asFloat()) / (*(*bottom).asFloat()); }
        /* addition */
                InternalNumber *operator+(InternalNumber& y)
                        { if (y.generality() == 20)
                                { Fraction *f = (Fraction *) &y;
                                 return new Fraction(
                                        *(*top * *f→bottom) + *(*f→top * *bottom),
                                                *bottom * *f→bottom);}
                         else {
                                 return InternalNumber::operator+(y);
                                }}
        /* division */
                InternalNumber *operator/(InternalNumber& y)
                        { if (y.generality() == 20)
                                { Fraction *f = (Fraction *) &y;
                                 return new Fraction( *top * *f→bottom,
                                                *bottom * *f→top);}
                         else {
                                 return InternalNumber::operator/(y);
```

```
                        }}
        /* multiplication */
                InternalNumber *operator*(InternalNumber& y)
                        { if (y.generality() == 20)
                                { Fraction *f = (Fraction *) &y;
                                 return new Fraction( *top * *f→top,
                                                *bottom * *f→bottom);}
                         else {
                                 return InternalNumber::operator*(y);
                                }}
        /* subtraction */
                InternalNumber *operator-(InternalNumber& y)
                        { if (y.generality() == 20)
                                { Fraction *f = (Fraction *) &y;
                                 return new Fraction(
                                        *(*top * *f→bottom) - *(*f→top * *bottom),
                                                *bottom + *f→bottom);}
                         else {
                                 return InternalNumber::operator-(y);
                                }}
        /* printing */
                void streamon(ostream& s)
                        { s << "Fraction " ; (*top).streamon(s);
                         s << "/"; (*bottom).streamon(s);}
                };

class Integer : public InternalNumber {
                int val;
                int generality() { return 10;}
        public:
                Integer(int i) { val = i;}

        /* printing */
                void streamon(ostream& s) { s << "Integer " << val; }

        /* addition */
                InternalNumber *operator+(InternalNumber& y)
                        { if (y.generality() == 10)
                                { Integer *i = (Integer *) &y;
                                  return new Integer(val + i→val);}
                         else {
                                 return InternalNumber::operator+(y);
                                }}

        /* conversion */
                InternalNumber *asFloat()
                        { return (new Float((double) val));}

        /* division */
                InternalNumber *operator/(InternalNumber& y)
                        { if (y.generality() == 10)
                                  return new Fraction(this, &y);
                         else {
                                 return InternalNumber::operator/(y);
                                }}

        /* multiplication */
                InternalNumber *operator*(InternalNumber& y)
                        { if (y.generality() == 10)
                                { Integer *i = (Integer *) &y;
                                  return new Integer(val * i→val);}
                         else {
                                 return InternalNumber::operator*(y);
                                }}
```

```
        /* subtraction */
                InternalNumber *operator-(InternalNumber& y)
                        { if (y.generality() == 10)
                                { Integer *i = (Integer *) &y;
                                  return new Integer(val - i->val);}
                         else {
                                  return InternalNumber::operator-(y);
                                }}

                };

class Number {
                InternalNumber *val;
                 Number() { delete(val); }

        public:
        /* construction */
                Number() { val = new InternalNumber();}
                Number(int i) { val = new Integer(i);}
                Number(double d) { val = new Float(d);}
                Number(InternalNumber *v) { val = v;}

        /* assignment */
                Number operator=(int i)
                        {          delete val; val = new Integer(i); return *this;}
                Number operator=(Number y)
                        {         val = y.val; return *this; }
                Number operator=(Number *y)
                        {         val = y->val; return *this; }

        /* arithmetic */
                friend Number operator+(Number&x, Number& y);
                friend Number operator-(Number&x, Number& y);
                friend Number operator*(Number&x, Number& y);
                friend Number operator/(Number&x, Number& y);

        /* printing */
                void streamon(ostream& s)
                        {         (*val).streamon(s); }
        };

/* friend functions */
inline Number operator+(Number&x, Number& y)
                { return *(new Number (*(x.val) + *(y.val))); }
inline Number operator-(Number&x, Number& y)
                { return *(new Number (*(x.val) - *(y.val))); }
inline Number operator*(Number&x, Number& y)
                { return *(new Number (*(x.val) * *(y.val))); }
inline Number operator/(Number&x, Number& y)
                { return *(new Number (*(x.val) / *(y.val))); }
inline ostream& operator<<(ostream&s, Number& n)
                { n.streamon(s); return s; }
```

Appendix 2: The Double Polymorphism Code

```
/*
        generalized arithmetic using double polymorphism
        written by Tim Budd
        Oregon State University
        Corvallis, OR 97331
/

# include <stream.h>

class InternalNumber {
        public:
        /* printing */
                virtual void streamon(ostream& s) { s << " undefined "; }

        /* addition */
                virtual InternalNumber *operator+(InternalNumber& y) ;
                virtual InternalNumber *addToFloat(InternalNumber& y) ;
                virtual InternalNumber *addToFraction(InternalNumber& y) ;
                virtual InternalNumber *addToInteger(InternalNumber& y) ;

        /* conversion */
                virtual InternalNumber *asFloat() ;
                virtual InternalNumber *asFraction() ;

        /* division */
                virtual InternalNumber *operator/(InternalNumber& y) ;
                virtual InternalNumber *divFromFloat(InternalNumber& y) ;
                virtual InternalNumber *divFromFraction(InternalNumber& y) ;
                virtual InternalNumber *divFromInteger(InternalNumber& y) ;

        /* multiplication */
                virtual InternalNumber *operator*(InternalNumber& y) ;
                virtual InternalNumber *multByFloat(InternalNumber& y) ;
                virtual InternalNumber *multByFraction(InternalNumber& y) ;
                virtual InternalNumber *multByInteger(InternalNumber& y) ;

        /* subtraction */
                virtual InternalNumber *operator-(InternalNumber& y) ;
                virtual InternalNumber *subFromFloat(InternalNumber& y) ;
                virtual InternalNumber *subFromFraction(InternalNumber& y) ;
                virtual InternalNumber *subFromInteger(InternalNumber& y) ;

                };

class Float : public InternalNumber {
                double val;
        public:
                Float(double d) { val = d; }

        /* printing */
                void streamon(ostream& s) { s << " Float " << val; }

        /* addition */
                InternalNumber *operator+(InternalNumber& y)
                        { return y.addToFloat(*this);}
                InternalNumber *addToFloat(InternalNumber& y)
                        {Float *f = (Float *) &y;
                         return new Float(val + f->val);}
                InternalNumber *addToFraction(InternalNumber& y)
                        {return addToFloat(*y.asFloat());}
                InternalNumber *addToInteger(InternalNumber& y)
                        {return addToFloat(*y.asFloat());}
```

```
        /* division */
                InternalNumber *operator/(InternalNumber& y)
                        { return y.divFromFloat(*this);}
                InternalNumber *divFromFloat(InternalNumber& y)
                        {Float *f = (Float *) &y;
                         return new Float(val / f->val);}
                InternalNumber *divFromFraction(InternalNumber& y)
                        {return divFromFloat(*y.asFloat());}
                InternalNumber *divFromInteger(InternalNumber& y)
                        {return divFromFloat(*y.asFloat());}

        /* multiplication */
                InternalNumber *operator*(InternalNumber& y)
                        { return y.multByFloat(*this);}
                InternalNumber *multByFloat(InternalNumber& y)
                        {Float *f = (Float *) &y;
                         return new Float(val + f->val);}
                InternalNumber *multByFraction(InternalNumber& y)
                        {return multByFloat(*y.asFloat());}
                InternalNumber *multByInteger(InternalNumber& y)
                        {return multByFloat(*y.asFloat());}

        /* subtraction */
                InternalNumber *operator-(InternalNumber& y)
                        { return y.subFromFloat(*this);}
                InternalNumber *subFromFloat(InternalNumber& y)
                        {Float *f = (Float *) &y;
                         return new Float(val - f->val);}
                InternalNumber *subFromFraction(InternalNumber& y)
                        {return subFromFloat(*y.asFloat());}
                InternalNumber *subFromInteger(InternalNumber& y)
                        {return subFromFloat(*y.asFloat());}
                };

class Fraction : public InternalNumber {
                InternalNumber *top, *bottom;
        public:
                Fraction(InternalNumber *t, InternalNumber *b)
                        { top = t; bottom = b; }

        /* fraction specific */
                InternalNumber *reciprocal()
                        { return new Fraction(bottom, top); }

        /* addition */
                InternalNumber *operator+(InternalNumber& y)
                        { return y.addToFraction(*this);}
                InternalNumber *addToFloat(InternalNumber& y)
                        { return (*asFloat()).addToFloat(y);}
                InternalNumber *addToFraction(InternalNumber& y)
                        { Fraction *f = (Fraction *) &y;
                         return new Fraction( *top + *f->top, *bottom + *f->bottom);}
                InternalNumber *addToInteger(InternalNumber& y)
                        { return addToFraction(*y.asFraction());}

        /* conversion */
                InternalNumber *asFloat()
                        { return (*(*top).asFloat()) / (*(*bottom).asFloat()); }

        /* division */
                InternalNumber *operator/(InternalNumber& y)
                        { return y.divFromFraction(*this);}
                InternalNumber *divFromFloat(InternalNumber& y)
                        { return (*asFloat()).divFromFloat(y);}
                InternalNumber *divFromFraction(InternalNumber& y)
```

```
                { return (*reciprocal()).multByFraction(y);}
        InternalNumber *divFromInteger(InternalNumber& y)
                { return new Fraction ( top , *bottom * y); }

    /* multiplication */
        InternalNumber *operator*(InternalNumber& y)
                { return y.multByFraction(*this);}
        InternalNumber *multByFloat(InternalNumber& y)
                { return (*asFloat()).multByFloat(y);}
        InternalNumber *multByFraction(InternalNumber& y)
                { Fraction *f = (Fraction *) &y;
                  return new Fraction( *top * *f→top, *bottom * *f→bottom);}
        InternalNumber *multByInteger(InternalNumber& y)
                { return new Fraction ( *top * y , bottom); }

    /* printing */
        void streamon(ostream& s)
                { s << "Fraction " ; (*top).streamon(s);
                  s << "/"; (*bottom).streamon(s);}
        };

class Integer : public InternalNumber {
        int val;
    public:
        Integer(int i) { val = i;}

    /* printing */
        void streamon(ostream& s) { s << "Integer " << val; }

    /* addition */
        InternalNumber *operator+(InternalNumber& y)
                { return y.addToInteger(*this);}
        InternalNumber *addToFloat(InternalNumber& y)
                { return (*asFloat()).addToFloat(y);}
        InternalNumber *addToFraction(InternalNumber& y)
                { return (*asFraction()).addToFraction(y);}
        InternalNumber *addToInteger(InternalNumber& y)
                { Integer *i = (Integer *) &y;
                  return new Integer(val + i→val);}

    /* conversion */
        InternalNumber *asFloat()
                { return (new Float((double) val));}

    /* division */
        InternalNumber *operator/(InternalNumber& y)
                { return y.divFromInteger(*this);}
        InternalNumber *divFromFloat(InternalNumber& y)
                { return (*asFloat()).divFromFloat(y);}
        InternalNumber *divFromFraction(InternalNumber& y)
                { return (*asFraction()).divFromFraction(y);}
        InternalNumber *divFromInteger(InternalNumber& y)
                { return new Fraction(this, &y); }

    /* multiplication */
        InternalNumber *operator*(InternalNumber& y)
                { return y.multByInteger(*this);}
        InternalNumber *multByFloat(InternalNumber& y)
                { return (*asFloat()).multByFloat(y);}
        InternalNumber *multByFraction(InternalNumber& y)
                { return (*asFraction()).multByFraction(y);}
        InternalNumber *multByInteger(InternalNumber& y)
                { Integer *i = (Integer *) &y;
                  return new Integer(val * i→val);}
```

```
        /* subtraction */
                InternalNumber *operator-(InternalNumber& y)
                        { return y.subFromInteger(*this);}
                InternalNumber *subFromFloat(InternalNumber& y)
                        { return (*asFloat()).subFromFloat(y);}
                InternalNumber *subFromFraction(InternalNumber& y)
                        { return (*asFraction()).subFromFraction(y);}
                InternalNumber *subFromInteger(InternalNumber& y)
                        { Integer *i = (Integer *) &y;
                         return new Integer(val - i->val);}
                };

class Number {
                InternalNumber *val;
        public:
        /* construction */
                Number() { val = new InternalNumber();}
                Number(int i) { val = new Integer(i);}
                Number(double d) { val = new Float(d);}
                Number(InternalNumber *v) { val = v;}

        /* assignment */
                Number operator=(int i)
                        {          delete val; val = new Integer(i); return *this;}
                Number operator=(Number y)
                        {          val = y.val; return *this; }
                Number operator=(Number *y)
                        {          val = y->val; return *this; }

        /* arithmetic */
                friend Number operator+(Number&x, Number& y);
                friend Number operator-(Number&x, Number& y);
                friend Number operator*(Number&x, Number& y);
                friend Number operator/(Number&x, Number& y);

        /* printing */
                void streamon(ostream& s)
                        {          (*val).streamon(s); }
        };

/* friend functions */
Number operator+(Number&x, Number& y)
                { return *(new Number (*(x.val) + *(y.val))); }
Number operator-(Number&x, Number& y)
                { return *(new Number (*(x.val) - *(y.val))); }
Number operator*(Number&x, Number& y)
                { return *(new Number (*(x.val) * *(y.val))); }
Number operator/(Number&x, Number& y)
                { return *(new Number (*(x.val) / *(y.val))); }
ostream& operator<<(ostream&s, Number& n)
                { n.streamon(s); return s; }
```