Improving C++ Performance Using Temporaries

Some programmers shy away from object-oriented languages because the resulting programs require too much overhead space and time at runtime. A strategy based on reusing temporary objects eliminates these inefficiencies and has a dramatic effect on the performance of OO programs.

Adair Dingle
Seattle University

Thomas H. Hildebrandt
Applied Microsystems Inc.

Object-oriented programming centers on the concept of an abstract data type, which localizes operations applied to a given data structure and provides a consistent interface to data structures having a known state. Thus, object-oriented programs are simpler to implement and maintain than those using traditional programming methods.

At the same time, object-oriented programming involves the creation and destruction of objects, which incurs the overhead costs of allocating and deallocating memory as well as initializing the object in a consistent state. Moreover, object-oriented programs can return fully constructed objects, causing unnamed temporary objects of the same type to be created in the scope of the calling routine. Both of these factors affect the performance of object-oriented programs compared to procedural programs.

For these reasons, programmers view object-oriented programming as wasteful compared to procedural programming. When runtime efficiency is important, developers have a legitimate reason to reject OOP.

Although functions that return objects can be avoided altogether—using return-by-reference semantics—this approach raises the issue of object ownership and aliasing problems. It also restricts the interface to an object, and so runs counter to the object-oriented principle of simplifying programming.

Instead of restricting the interface, we propose to improve the efficiency of the underlying implementation by reusing temporaries. Our experimental results show large speedups using this method.

ROLE OF TEMPORARIES

A function written in an object-oriented language creates a return value by constructing an object with a return-value type and passing this object back to the calling routine.[1] The returned object has no name, and it can be accessed only as an operand. Such a return-

0018-9162/98/$10.00 © 1998 IEEE

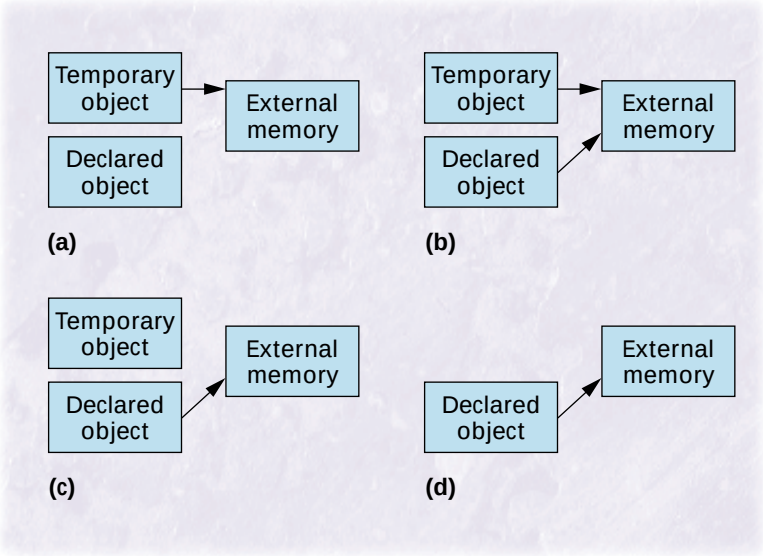

Figure 1. In copy-by-assumption, copying the pointer transfers the memory owned by the operand to the target object. Copying even a relatively small object by assumption takes significantly less runtime than making a verbatim copy. (a) Before function return; (b) assignment after return; (c) temporary releases memory; (d) memory outlives temporary.

value object is typically destroyed at the end of the statement in which it was constructed. Because its lifetime is shorter than that of an object explicitly declared in the same routine, it is also known as a *temporary return-value object*, or a *temporary*.

For example, in `A = B + C`, the expression `B + C` returns a temporary that is then used as the operand of the assignment operator (=). The assignment operator typically places a verbatim copy of its operand into the target object, possibly returning a reference to the target object as well. The temporary is then destroyed.

Reusing temporaries

Creating and destroying a temporary involves allocating the memory required to hold it and then deallocating the same memory a short time later. The overhead of initializing the object and copying out the useful information before it is destroyed also takes memory space. On the surface, then, it would seem reasonable to create as few temporaries as possible, but this runs counter to the aim of object-oriented programming, which is to simplify.

Others have devised ways to use temporaries in functional languages,[2-4] but both published methods require the manipulation of object code. This approach is practical for functional languages because they place a severe restriction on coding style—namely, they prohibit the declaration of variables. In C++, in contrast, the complete lack of restrictions on the semantics of functions and operators makes it nearly impossible to optimize object management at the object-code level.

Assignment-by-assumption

In the case of object-oriented languages, one known method—the *assignment-by-assumption* technique[5]—eliminates unnecessary verbatim copies of the temporary object when it becomes the operand of an assignment operation. In this technique, the target *assumes* the temporary: The memory is transferred from the temporary to the target object by copying a data pointer from one object to the other, after deleting the former contents of the target object.

This method is similar to reuse in functional languages, but because object-oriented languages impose fewer restrictions on programming style, the method must consider additional object storage classes to ensure correctness of the resulting code.

In addition, assignment-by-assumption does not rely on access to the object code—it can be implemented entirely in source code through inheritance, operator overloading, redefinable constructors and destructors, and assignment operators.[6] Whether implemented at the source or object code level, assignment-by-assumption represents a significant innovation.

COPY-BY-ASSUMPTION

We have generalized the assignment-by-assumption method to create what we call *copy-by-assumption*, whereby any function can potentially reuse the memory allocated to a temporary object. In chained operations, a temporary object can become an operand of a function other than the assignment operator. In order to maximize the efficiency gains suggested by assignment-by-assumption, it is thus necessary to allow arbitrary functions to reuse temporary operands.

The usual semantics of the assignment operator are to replace the target object with a verbatim copy of the operand and to return a reference copy of the (updated) target. A verbatim copy is distinct from the original: A change to the copy does not alter the original. In contrast, a change to a reference copy (or *alias*) also changes the original.

When the operand of an assignment operation is a declared object, as in `A = B`, a verbatim copy operation is required. However, if the operand is a temporary, as in `A = temp1` (where `temp1 = B + C`), we can use *copy-by-assumption*.

In copy-by-assumption, copying the pointer transfers the memory owned by the operand to the target object, as Figure 1 shows. Copying even a relatively small object by assumption takes significantly less runtime than making a verbatim copy.

```
class TmpVector;        // Forward declaration
class Vector;           // Forward declaration
class RefVector
{
   unsigned len;        // Records the length of the vector
   double* array;       // Stores the dynamically allocated data
 public:
   RefVector operator=(RefVector&);          // Assignment (case 1)
   friend TmpVector operator+(RefVector&, RefVector&);
};

class Vector:public RefVector
{
 public:
   RefVector operator=(RefVector&);          // Assignment (case 2)
   RefVector operator=(TmpVector&);          // Assignment (case 3)
};

class TmpVector:public Vector
{
 public:
   TmpVector operator=(RefVector&);          // Assignment (case 4)
   TmpVector operator=(TmpVector&);          // Assignment (case 5)
};
```

Figure 2. Interface specification for a `Vector` class that implements dynamically allocated vectors of double-precision numbers.

Data pointed to by a temporary object is transferred to a target object in three steps:

1. Release any data currently pointed to by the target object.
2. Copy the data pointer in the temporary object to the data pointer in the target object.
3. Set the data pointer in the temporary object to the null pointer.

OO considerations

Implementing assignment correctly involves considering two cases: one in which the operand is defined as a declared object and one in which it is defined as a temporary. (We define declared and temporary objects as distinct types, to enable operator overloading.) The compiler then selects the appropriate assignment method on the basis of the operand type. This assumes that the target object is a declared object. In addition, we must consider two parallel cases in which the target object is a temporary.

It is also possible to define a third type that acts like a reference copy. For example, the compiler represents the chained assignment expression `A = B = C` as two separate assignments:

```
temp1 :=  B = C;
temp2 :=  A = temp1;
```

Here, `temp1` is the value returned by the assignment of `C` to `B`, and `temp2` is the value returned by the assignment of `temp1` to `A`. While the return value must be available for chained assignment, it should also be inexpensive to construct, in case it is not used. In this example `temp1` is used but `temp2` is not. When a reference copy is the operand of an assignment, the assignment operation must be implemented as a verbatim copy. In this way, the chained assignment can be implemented efficiently.

Using inheritance

Including a reference object in the abstract data type increases the number of cases that the copy-by-assignment technique must consider. Now instead of four there are nine. Without inheritance, including a reference object would have to be implemented as nine overloaded assignment operators. However, by inheriting the target subtype from the reference subtype and the temporary subtype from the target subtype, the number of cases to consider is reduced to five.

The C++ code in Figure 2 illustrates the original

```
RefVector RefVector::operator=(RefVector& R)
 // An assignment operator that takes a vector as its operand,
 // assigns it to a RefVector, and returns a Ref copy of the result
{
 if (len != R.len)                        // Make sure lengths match
   throw VectorLengthMismatch();
 for (unsigned i = 0; i < len; I++)       // Verbatim copy
   array[i] = R.array[i];
 return *this,                            // Return Ref copy
}
```

Figure 3. Implementing the assignment to a `RefVector` requires first that the length of the target object match the operand, then that a verbatim copy be made and also returned as a reference copy of the target object. This implementation overwrites the underlying `Vector` or `TmpVector` with the data to which the `RefVector` operand points.

```
RefVector Vector::operator=(RefVector& R)
 // Normal case: The operand is a RefVector or Vector
 // Make target an exact copy of the operand, using a verbatim copy
{
   if (len != R.len)
   {
     delete [] array;                          // Adjust length of target
     array = new double[len = R.len];
   }
   for (unsigned i = 0; i < len; i++)          // Verbatim copy
   array[i] = R.array[i];
   return *this;                               // Return a Ref copy
};

RefVector Vector::operator=(TmpVector& T)
 // Special case: The operand is a TmpVector
 // Use copy-by-assumption
{
 delete [] array;     // Deallocate current contents
 len = T.len;
 array = T.array;     // Assume data from T
 T.array = 0;         // (pointer copy)
 return *this;        // Return a Ref copy of "this"
};
```

(a)

```
TmpVector TmpVector::operator=(RefVector& R)
  // Normal case: A RefVector or Vector as the operand
  // Make target an exact copy of the operand
{
 Vector::operator=(R);    // Use Vector::operator=(RefVector&)
 return *this;            // Cast result as TmpVector
};

TmpVector TmpVector::operator=(TmpVector& T)
  // Special case: Assignment of a TmpVector to a Vector
  // Use copy-by-assumption
{
   Vector::operator=(T); // Use Vector::operator=(TmpVector&)
   return *this;         // Cast result as a TmpVector
};
```

(b)

Figure 4. Implementation of assignment to both (a) `Vector` and (b) `TmpVector`.

assignment-by-assumption technique.[5] The `Vector` class implements dynamically allocated vectors of double-precision numbers; Figure 2 shows its interface specification. Within each class declaration, parameters of type `RefVector&` represent (through inheritance) references to both `RefVectors` and `Vectors`, thus eliminating three cases. Within the declaration for the `RefVector` class, `RefVector&` parameters also represent `TmpVectors`. Altogether, inheritance eliminates four of the original nine cases.

Three of the assignment operators have the same signature: They accept a reference to a `Vector` and return a `RefVector`. However, in the `TmpVector` class the return value is a `TmpVector`. This is done so an *enclosing* operator or function can assume the result of the assignment. Thus, a statement such as `A = (B + C) = D + E` will execute efficiently, even though the statement itself is an inefficient way of coding `A = D + E`.

As Figure 3 shows, assignment to a `RefVector` requires that the length of the target object match that of the operand. It also requires a verbatim copy, thus overwriting the underlying `Vector` or `TmpVector` with the data to which the `RefVector` operand points.

Figure 4a shows the implementation of the assignment to a `Vector` and Figure 4b shows the assignment to a `TmpVector`. In both implementations, the return statement calls a *copy-constructor* to convert the type of its operand to the specified return-value type. In the case of the `TmpVector`, the two returns call the copy-constructor `TmpVector(TmpVector&)`. Because a `TmpVector` can only be used as an operand, it makes no sense to leave data attached to the `TmpVector` target of these two assignment operators. Thus, the `TmpVector (TmpVector&)` copy-constructor uses copy-by-assumption.

IMPROVING ASSIGNMENT PERFORMANCE

Let us examine in detail how reusing temporaries in assignment operations improves performance, using the two variable-length vectors defined above. The C++ code to add variable-length vectors is:

```
Vector A, B, C;
 . . .     // A and B are initialized
C = A + B;
```

Figure 5 diagrams three ways that the allocation, deallocation, and write operations might be performed on the objects involved in the statement `C = A + B`.

In all three approaches, the calling routine creates the source objects `A` and `B` and the destination object `C`. Next, the program invokes the addition operation. This routine binds the actual parameters `A` and `B` to the formal parameters `P` and `Q`, respectively, using call-by-reference. It then creates a new object, `R`, to hold the result. The routine performs an element-by-element addition and writes the results into `R`.

In the implementations shown in Figure 5a and 5b, as the addition operation completes, the value in `R` is copied into an unnamed temporary object using the copy-constructor, and `R` is deallocated immediately.

This unnamed temporary is then supplied as the operand to the assignment operation and hence becomes bound to the formal parameter `T`. The target of the assignment operation, `C`, is bound to the formal parameter `S`. The assignment operation overwrites `S` (and `C`) with the elements stored in `T`. In the calling routine, the unnamed temporary is deallocated at the end of the statement, and the named variables are deallocated at the end of the routine.

The implementation in Figure 5a does not reuse temporaries and as a result requires five allocations and three write operations.

Eliminating verbatim copies

The implementation in Figure 5b does use temporaries (via copy-by-assumption) to eliminate two verbatim copy operations:

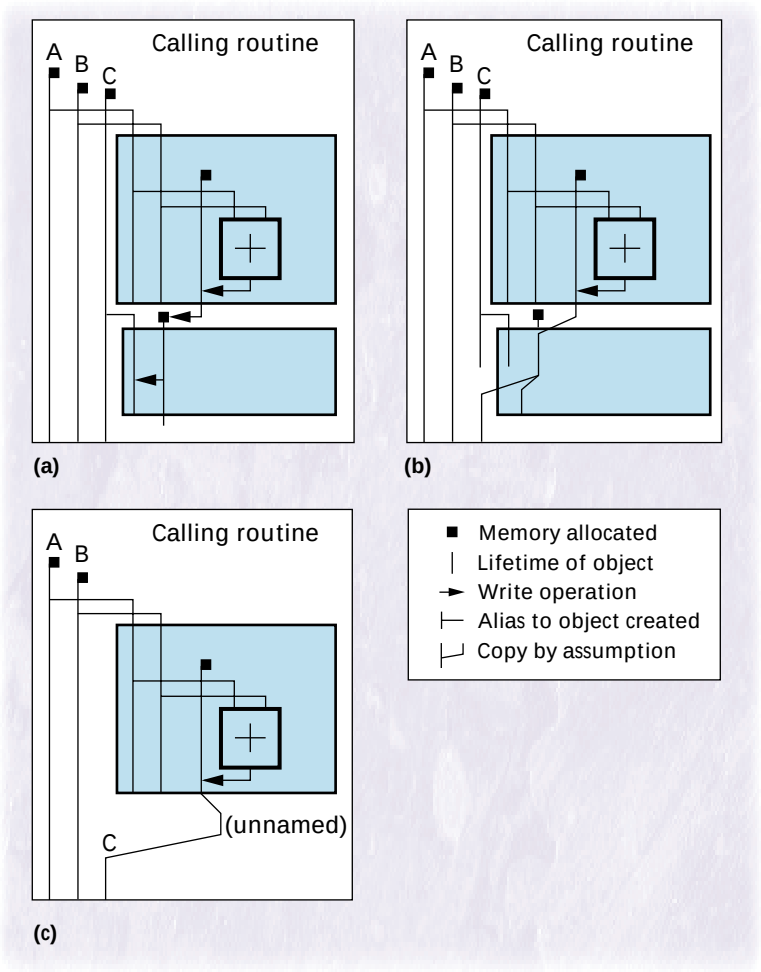

Figure 5. Diagrams of memory allocation, deallocation, and write operations performed on the objects involved in the statement `C = A + B`. (a) A program that uses no temporaries; (b) temporaries reused (copy-by-assumption); and (c) assumption-initializer used.

```
TmpVector operator+(RefVector& A, RefVector& B)
     // This is the "standard" implementation
{
     if (A.len != B.len)                // Make sure lengths match
       throw VectorLengthMismatch();
     TmpVector C = A;                   // Uses a verbatim copy
     for (unsigned i = 0; i < A.len; i++)
       C.array[i] += B.array[i];
     return C;                          // Uses copy-by-assumption
};
```

(a)

```
TmpVector operator+(RefVector& A, TmpVector& B)
     // This version "assumes" data from B and then adds in A
{
     if (A.len != B.len)                // Make sure lengths match
       throw VectorLengthMismatch();
     TmpVector C = B;                   // Uses copy-by-assumption
     for (unsigned i = 0; i < A.len; i++)
       C.array[i] += A.array[i];
     return C;                          // Uses copy-by-assumption
};
```

(b)

Figure 6. Two nondestructive addition subroutines: (a) uses a verbatim copy operation while (b) uses the faster copy-by-assumption operation.

- Note that the diagram is identical to Figure 5a up to the point at which the addition operator is exited. At this point, instead of performing a verbatim copy from `R` to the unnamed temporary, the unnamed temporary assumes `R`'s memory, thus eliminating one verbatim copy.
- The other verbatim copy is eliminated within the assignment operator. As in Figure 5a, the unnamed temporary is supplied as an operand of the assignment operator and is bound to the formal parameter `T`. The target of the assignment operator, `C`, is bound to the formal parameter `S`. The assignment operation is performed using copy-by-assumption: It deallocates the memory belonging to `S`, and `S` then assumes the memory that belonged to `T`. Because `S` is an alias for `C`, this assumption effectively replaces `C`'s memory with the contents of the unnamed temporary. Hence, this method requires five allocations and one write.

Preventing allocations altogether

Replacing the assignment operation with an initializer achieves even greater performance gains, as illustrated in Figure 5c. Recall that an unnamed temporary is created as the addition operation completes. Because the memory allocated to this temporary is immediately replaced with the memory allocated to `R`, it need never be allocated in the first place.

In other words, when the operand of an assignment is a temporary and the target object has not yet been initialized, the copy-constructor can replace the assignment operator. The new object is initialized using copy-by-assumption.

We call this form of the copy-constructor the *assumption-initializer*. The calling routine that uses the assumption-initializer is written:

```
Vector A, B;
 . . .              //A and B are
                    //initialized
Vector C(A + B); //C assumes
                    //temporary's memory
```

In this case the required number of allocations drops from five to three, as in Figure 5c. Thus we can save two writes and two allocations overall.

IMPROVING PERFORMANCE OF OTHER FUNCTIONS

The original paper outlining this reuse technique focused on reusing temporaries in assignment operators only.[5] But the technique can be generalized to other operators and functions: Every time you can eliminate an allocation/deallocation pair by reusing a temporary, you improve performance.

```
class TmpVector;                        // Forward declaration
class Vector
{
   friend class TmpVector;
   int size;                            // Memory internal to the object
   int alloc;
   double* data;                        // Memory external to the object

public:
   Vector();                            // Sample constructors
   Vector(int Alloc);
   Vector(const Vector&);
   Vector(TmpVector&);
   ~Vector();                           // Destructor

   void operator=(const Vector&);       // Assignment operator
   void operator=(TmpVector&);
    // Sample nondestructive operator
   friend TmpVector operator+(const Vector&,const Vector&);
   friend TmpVector operator+(TmpVector&, const Vector&);
   friend TmpVector operator+(const Vector&, TmpVector&);
   friend TmpVector operator+(TmpVector&, TmpVector&);
    // Sample destructive operator
   void operator+=(const Vector&);
};
```

Figure 7. Declaration of `Vector` class to reuse temporaries. It uses two declarations for the assignment operand, one each for `Vector` and `TmpVector` operands, and four declarations of the addition operator to account for the possible arrangements of `TmpVector` operands.

Such pairs are potentially created whenever a function returns a value. This necessarily creates at least one object—the return-value object—and may also create local objects to store intermediate results. In either case, memory is allocated to hold the object, and external memory may also be allocated by the object. Yet both the object and its external memory will only be deallocated later. If the program can instead assume the external memory from a temporary supplied as an operand, it can save an allocation/deallocation pair.

Using copy-by-assumption in operations besides assignment involves reassigning the data attached to a `Tmp` operand to another object of the same abstract type. Because this data need not be preserved once it has been transferred, whenever an operand has a `Tmp` subtype, nondestructive operators can be rewritten using destructive operators.

For example, although the two subroutines in Figure 6 look almost identical, they differ in their constructor calls. The initialization `TmpVector C = A` calls the constructor `TmpVector(RefVector&)`, which is implemented using a verbatim copy operation. In contrast, the initialization `TmpVector C = B` calls the constructor `TmpVector(TmpVector&)`, which uses the faster copy-by-assumption operation.

SAMPLE IMPLEMENTATION

Let's suppose you wanted to reuse temporaries in a C++ program that implements a `Vector` type. We will use the sample code in Figure 7 to illustrate how to do this.

The program first defines a subtype that represents a declared object, `Vector`, and then inherits the `TmpVector` class from the base class. (In this explanation we include only the `Vector` and `TmpVector` subclasses.)

Objects in both `Vector` and `TmpVector` behave identically *except* when reuse occurs. A formal parameter of the `Tmp` type in a function header indicates that the corresponding object can be reused in the function body after it is last accessed. When the memory in a temporary has been reused, its memory pointer is set to zero, signaling that it no longer has any accessible memory associated with it. The destructor must ensure that no action is taken when an attempt is made to deallocate a null pointer.

In the case of a function definition with one or more temporaries among its formal parameters, the program can reuse the memory attached to these temporaries after all pertinent information is extracted from them. This can be accomplished using either

Reusing temporaries does not increase the size of the executable image unreasonably.

assumption-initialization or copy-by-assumption.

In the case of an abstract data type, reuse can be accomplished by including overloaded versions of each function for every possible combination of target and temporary operands. But this leads to a proliferation of cases and the attendant source code expansion. Instead, the programmer can exploit symmetries—actually implementing only one case and using inline functions to cover the rest—so that only those cases of temporary operands that are actually used will appear in the object code.

Inheritance further reduces code size: Recall that the `Tmp` subtype is inherited from the base class. You need code only those cases in which the behavior of a `Tmp` object differs from that of a declared object, and this arises only in constructors and the assignment operator. Reusing temporaries, therefore, does not increase the size of the executable image unreasonably.

Class declarations

The declaration of the `Vector` class in Figure 7 has two declarations for the assignment operator: one with a `Vector` operand and one with a `TmpVector` operand. And it has four declarations for the addition operator to account for four possibilities. There is one possibility that there will be no `TmpVectors` in the operand list. There are two possibilities that there will be one `TmpVector`. And there is one possibility that there will be two.

The declaration of the `TmpVector` class is:

```
class TmpVector : public Vector
{
 public:
   TmpVector();    //Constructors
   TmpVector(int Alloc);
   TmpVector(TmpVector& V);

   ~TmpVector();   //Destructor
};
```

There are just two constructors: the default constructor and the copy-constructor. Other than this, the functions in the `TmpVector` behave as inherited from `Vector`; they need not be redefined.

There are some member functions that do differ in the type of the target object, as we describe next.

Constructors

The implementations of the constructors for the `Vector` and `TmpVector` classes are shown in pairs below, so that their differences can be noted easily.

Default. The `Vector` default constructor is

```
Vector::Vector()
{
 size = 0;
 alloc = 0;
 data = 0;
}
```

Memory allocation is set to zero, since this value later helps determine if external memory is attached to a `Vector` object.

The `TmpVector` default constructor is

```
TmpVector::TmpVector() : Vector() {}
```

This constructor behaves identically, but C++ constructors cannot be inherited. Hence this constructor is necessary to simply invoke the `Vector` constructor.

Preallocation. Preallocation constructors—which allocate space, but return a `Vector` or `TmpVector` of length zero—function identically. In `Vector` the definition is

```
Vector::Vector(int Size)
{
 size = Size;
 alloc = size;
 data = new double[alloc];
};
```

While the `TmpVector` definition is

```
TmpVector::TmpVector(int Size) :
   Vector(Size) {}
```

Copy. The copy-constructor definition in `Vector` is

```
Vector::Vector(const Vector& V)
{
   size = V.size;
   alloc = size;
   data = new double[alloc];
   Vector::operator=(V);
}

Vector::Vector(TmpVector& V)
{
   size = V.size;
   alloc = V.alloc;
   data = V.data;
   V.data = 0;
}
```

where `Vector::Vector(const Vector& V)`

represents the copy-constructor and `Vector::Vector(TmpVector& V)` the assumption initializer.

The `TmpVector` definition is

```
TmpVector::TmpVector(TmpVector &V)
   : Vector(V) {}
```

In these examples, the form `Vector(Vector&)` makes a verbatim copy of the operand while the form `Vector(TmpVector&)` merely assumes the data from its `TmpVector` operand. `TmpVector(TmpVector&)` is identical in function to `Vector(TmpVector&)`, except that the resulting object is cast to the `TmpVector` type.

Destructors

The destructor for `Vector` merely deletes the data array that has been allocated to it:

```
Vector::~Vector()
{
 delete [] data;
}
```

While the destructor for the `TmpVector` class calls `~Vector` implicitly.

```
TmpVector::~TmpVector() {}
```

Assignment

The implementations of the assignment operator illustrate the difference between a verbatim copy and a copy-by-assumption. In `Vector`, the overloaded operator copies the data values from the source to its own data field.

```
void Vector::operator=
   (const Vector& V)
{
   if (size != V.size)
     throw Vector_size_mismatch();
   int Size = size;
   double* out_ = data;
   double* in_ = V.data;
   while (Size-- > 0)
     *out_++ = *in_++;
}
```

In contrast, in `TmpVector` the operator deletes its own allocated data and then assumes the data memory of the source:

```
void Vector::operator=(TmpVector& V)
{
   delete data;
   size = V.size;
   data = V.data;
   alloc = V.alloc;
   V.data = 0;
}
```

```
TmpVector operator+(const Vector& A,
   const Vector& B)
{
   int Size = A.size <? B.size;
   TmpVector C(Size);     // Allocates new memory
   double* a_ = A.data;
   double* b_ = B.data;
   double* c_ = C.data;
   while (Size-- > 0)
     *c_++ = *a_++ + *b_++;
   return C;
}
TmpVector operator+(TmpVector& A,
   const Vector& B)
{
   TmpVector C(A);        // Data assumed from A
   C.Vector::operator+=(B);
   return C;
}

inline TmpVector operator+(const Vector& A,
   TmpVector& B)
{
   TmpVector C(B);      // Data assumed from B
   C.Vector::operator+=(A);
   return C;
}
inline TmpVector operator+(TmpVector& A,
   TmpVector& B)
{
   TmpVector C(A);      // Data assumed from A
   C.Vector::operator+=(B);
   return C;
}
```

(a)

```
void Vector::operator+=(const Vector& V)
{
   int Size = size <? V.size;
   double* in_ = V.data;
   double* out_ = data;
   while (Size-- > 0)
     *out_++ += *in_++;
}
```

(b)

Figure 8. Reuse in (a) nondestructive addition; (b) destructive addition (included for completeness).

Table 1. Execution times for various operations on 1,024-element vectors of double-precision numbers.

Operation	libg++ (μs)	`Temp Reuse`(μs)	Percent speedup*
Resizing	338	1.69	20,000
Multiplication	1,201	596	101
Concatenation	1,880	968	94
Reversal	907	485	87
Selection	951	527	80
Addition	1,038	589	76
Subtraction	1,038	593	75
Division	2,041	1,483	38
Assignment	591	506	17

* Speedup=[(TimeOld/TimeNew) – 1] × 100

Table 2. Execution times for various operations on 512-element character strings.

Operation	Rogue Wave(μs)	`Temp Reuse`(μs)	Percent speedup*
Resizing	2.216	0.240	823
In-Place Upcase	1,609	316	409
Transfer Upcase	1,822	365	399
Assignment	0.991	0.284	181
Selection	49.6	32.5	53
Concatenation	90	75	25

* Speedup=[(TimeOld/TimeNew) – 1] × 100

Nondestructive addition

We will use the addition operation to illustrate how reuse works in a nondestructive action. Figure 8a shows the implementation. In the two cases in which one `TmpVector` appears in the operand list, the addition operator uses assumption-initialization to create the return value (and in doing so reuses the `TmpVector` operand) and then adds the other operand using destructive addition. Figure 8b shows the destructive addition operator. In the one case in which two `TmpVectors` appear in the operand list, only one can be reused. In Figure 8a, we arbitrarily choose A.

EXPERIMENTAL RESULTS

We benchmarked our method of reusing temporaries against two established software packages—the Free Software Foundation's GNU libg++ and Rogue Wave's tools.h++ libraries—that eliminate copying temporaries under certain circumstances.

We ran a variety of operations on vectors of double-precision numbers and on character strings, and found that our method of reusing temporaries resulted in significant speedups. For both tests, we chose compiler switches that would maximize the execution speed and varied vector lengths by powers of two from 1 to 1,024.

GNU libg++

The GNU libg++ version 2.5.8 C++ library does not make significant use of temporaries, but does make use of a nonstandard language construct that lets programmers circumvent calls to a copy-constructor, usually generated by the return statement. For this reason, the speedup reported for this comparison is less than would have been obtained in comparison with a naive implementation.

We compared the arithmetic vector class supplied with the GNU libg++ library to a comparable arithmetic vector class (`Temp Reuse`) using temporary reuse in its implementation. We saw the greatest relative speed increases for larger vectors. For all operations except assignment, our implementation was faster even for the shortest vectors. We used a *dual loop technique* to obtain experimental timing data, which removed loop overhead from the calculations. We gathered a sufficient number of samples to guarantee a standard deviation of at most 1 percent. Table 1 shows the execution times and speed increase for 1,024-element vectors.

The astounding speedup for the resizing operation is due to the fact that libg++ makes no attempt to reuse previously allocated memory, even if the new size of the vector is smaller than its present size. The relative speedup for more expensive operations is lower. Division, which is computationally expensive, gave a smaller relative speedup than multiplication. However, the absolute speedup (in μs) is almost the same for both.

To provide a fair comparison, the measurement on assignment was for the case in which a verbatim copy was performed. In situations where our model could use copy-by-assumption, the relative speedup would increase steadily with increasing vector size. For 1,024-element vectors, the relative speedup in this case was 11,600 percent.

Rogue Wave's tools.h++

We compared operations on character strings between a commercial C++ library, Rogue Wave Software's tools.h++ (Version 6.0) and a comparable string class (`Temp Reuse`) using temporary reuse. Rogue Wave's library uses a copy-on-write scheme to avoid copying temporaries under certain conditions. However, the copy-on-write scheme is based on reference counting, which introduces some runtime performance penalties.[7]

The compiler used for both class implementations was Borland C++ Version 4.0. With the exception of substring selection, the relative speed increase stayed the same or increased as the vector length was increased. Table 2 shows the execution times and speed increases for 512-element strings.

In both libraries, the time required to assign or resize a string object was independent of the object size (up to 1,024 bytes in length). The discrepancy clearly shows the overhead costs in tools.h++ associated with initializing and maintaining a reference count.

Table 3. Time required, in μs, to concatenate two 25-character strings.**

Library (Compiler)	Without reuse	With reuse	Percent speedup*
GNU libg++ V2.5.3 (GNU g++ V2.5.8)	21.83	13.17	66
MFC V2.0 (Microsoft C/C++ V7.0)	19.80	12.10	64
Borland Object Windows (Borland C++ V4.0)	11.00	7.70	43
Rogue Wave tools.h++ (Borland C++ V4.0)	4.39	3.51	25

* Speedup=[(TimeOld/TimeNew) − 1] × 100

** Tests run on different machines—execution times comparable only within each example.

Other libraries

The observed performance improvement is not specific to the compiler and library used, as our tests using two other available C++ compilers and their associated libraries shows. These experiments compared the amount of real time required to concatenate two 25-character strings, with and without temporary reuse. In every case, reusing temporaries resulted in a significant performance gain, as shown in Table 3.

Experimental results show that reusing temporaries results in significant performance gains for a variety of operations, compared to available C++ libraries. Though cast in the framework of C++, the ideas presented here can find application in a variety of programming languages, even in languages that rely on garbage collection for memory management (such as CLOS and Java).

Because it deals closely with dynamic memory allocation, our method may also become the basis for an efficient memory management scheme. We are working on a code analyzer that will verify that the memory allocation, deallocation, and transfer rules of our method have been used correctly. Preliminary studies on a complete matrix analysis package incorporating these ideas show that our method can also consistently apply to modules that define and manipulate common data structures (as in linked lists, stacks, queues, and dynamic arrays). We plan to implement the Standard Template Library to demonstrate its utility. We believe that our method can yield even greater performance gains when applied at several levels of abstraction. ❖

References

1. B. Stroustrup, *The C++ Programming Language,* 2nd ed., Addison-Wesley, Reading, Mass., 1991, pp. 126-127.
2. I. Foster and W. Winsborough, "Copy Avoidance Through Compile-Time Analysis and Local Reuse," *Proc. Int'l Symp. Logic Programming,* MIT Press, Cambridge, Mass., 1991, pp. 455-469.
3. P. Schnorf, M. Ganapathi, and J.L. Hennessy, "Compile-Time Copy Elimination," *Software Practice and Experience,* Nov. 1993, pp. 1,175-1,200.
4. K. Gopinath and J.L. Hennessy, "Copy Elimination in Functional Languages," *Conf. Record 16th Ann. ACM Symp. Principles Programming Languages,* ACM Press, New York, 1989, pp. 303-314.
5. T.H. Hildebrandt, "How to Avoid Copying in Assignment Operations: A Class-Based Approach," *J. Object-Oriented Programming,* Sept. 1993, pp. 25-29.
6. T.H. Hildebrandt, *Method for Reusing Temporaries and Reclaiming Shared Memory,* US Patent # 5,535,390, Patent and Trademark Office, Washington, D.C., 1996.
7. J. Heymann, "A Comprehensive Analytical Model for Garbage Collection Algorithms," *SIGPlan Notices,* Vol. 26, No. 8, 1991, pp. 50-59.

Adair Dingle is an assistant professor of computer science at Seattle University. Her research interests include algorithms and practical and pedagogical aspects of object-oriented programming. She received a BS in mathematics from Duke University and an MS in computer science from Northwestern University. As a Texas Instruments Fellow, she received a PhD in computer science from the University of Texas at Dallas.

Thomas H. Hildebrandt is a senior software engineer at Applied Microsystems Inc. He received a BS in electrical engineering and computer science from the University of Colorado; and an MS in electrical and computer engineering and a PhD in computer engineering, both from North Carolina State University.

Contact Dingle at Computer Science and Software Engineering Dept., Seattle Univ., 900 Broadway, Seattle WA 98122-4460; dingle@seattleu.edu. Contact Hildebrandt at Applied Microsystems Inc., PO Box 97002, Redmond, WA 98073-9702; hilde@amc.com.