The Power of C++: The Making of TMON 3.0

Waldemar Horwat

Presented at MacHack '90

Abstract

For several years C and Pascal have been viewed as roughly equivalent Macintosh programming languages, differing in syntax but not expressiveness. This, however, is not the case with Object Pascal and C++; C++ is a fundamentally more expressive language than Object Pascal or C. I will demonstrate this by exploring C++ in depth along one particular dimension—defining seamless language extensions. These extensions include a complex number library, a file type, an unlimited-length string package, and a garbage collector. Similar extensions were used in the design of TMON 3.0.

As these extensions are explored in depth, it becomes apparent that they provide a substantial increase of reliability and decrease in debugging effort for complex programs. However, to reap these benefits, one must be familiar with C++ and willing to use it for an entire project.

1. Introduction

Programming Paradigms

Although few Macintosh programmers think about it much, most of them are using a certain, limited *programming paradigm* when writing software. The *procedural* paradigm is most frequently used on the Macintosh, as it is encouraged by the use of traditional programming languages such as Pascal, C, or assembly language (when used the “ordinary” way). Moreover, all of the interfaces in Inside Macintosh also follow the procedural style.

In the procedural paradigm, the programmer views the world as consisting of procedures and functions which call other procedures and functions, passing values and data structures as arguments. The programmer is burdened with maintaining every aspect of these data structures.

Although the procedural programming paradigm is a good one for many applications, it is by no means the only one, nor necessarily the best one for a particular application. At this time Prolog [10], Common Lisp [9], and many specialty languages offer different paradigms, often several combined in the same language. In contrast, several years ago Applesoft BASIC and its peers defined the dominant programming paradigm for beginning personal computer users.

Applesoft BASIC was several levels below the procedural programming paradigm, for it lacked good data structuring abilities such as Pascal's records or C's structures. Furthermore, Applesoft BASIC lacked procedures and functions. It was possible to GOSUB to a common piece of code, but this was more of a macro than a procedure—there were no provisions for local variables, and passing arguments was cumbersome at best. Finally, Applesoft BASIC was an unstructured language at a time when the *structured programming vs. GOTO* debate was winding down. Structured programming won the debate—at this time virtually every new programming language uses structured control flow constructs such as WHILE and REPEAT loops and IF/THEN/ELSE statements. Nevertheless, not so long ago in Applesoft BASIC, GOTOs were the main means for branching in a program.

I believe that in ten years we will look upon the current programming tools, languages, and paradigms from the same point of view as we now view Applesoft BASIC. I do not know which paradigm will emerge as the dominant one in the year 2000, but chances are it already exists today. Object-oriented programming is a good starting point, but it is definitely not

enough for tomorrow's software projects. Other paradigms, especially parallel ones such as the ones that I described in a paper presented at last year's MacHack [4], posses some advantages.

Development of C++

C++ [11] [12] is an object-oriented programming language developed by Bjarne Stroustrup based on C [5]. Two of the main goals of the development of C++ were to bring the level of the language up to the problem being solved and to ease development of large programs. C++ achieves these goals by introducing the object-oriented paradigm, along with other features, without sacrificing efficiency. The fact that C++ is as efficient as C may be the reason for C++'s wide acceptance where other object-oriented languages such as SmallTalk have not done as well.

C++ gradually evolved over the last decade. The latest revision of the language is version 2.0 [12], which will be the version used in this paper; this is also the version provided in the current MPW C++ package. C++ is still evolving, and it will likely change in the future.

The Power of C++

The complexity of C++ approaches that of Ada [1], and my goal here is to teach neither C++ nor object-oriented programming. Instead, I will explore the C++ language along one particular dimension that I found especially useful—defining increasingly more sophisticated “extensions” to the language. To ensure that these extensions are practical, I will use code similar to that found in TMON 3.0 for the examples that follow.

One view of programming holds that writing a program in a structured language consists of repeatedly adding extensions (procedures, functions, types, macros, etc.) to the base language until the extended language allows the problem to be solved in a single main procedure. Of course, some languages are better than others in facilitating extensions, and, as I will try to show, among the commonly used Macintosh compiled languages, C++ is probably the best.

This paper is intended for both intermediate and experienced C++ users and for C and Pascal programmers curious about C++. I hope that current C++ users will find the examples here useful and will expand their horizons about what the C++ language can do for them. I also hope that some of the programmers considering using C++ will find here reasons to learn and use C++. What I am about to describe simply cannot be done in any practical sense in Pascal, Object Pascal, or C.

2. Creating a New Type

The first extension example is simple: Extend the language to allow the use of complex numbers in arithmetic expressions. A complex number library can be written in Pascal, Object Pascal, C, or C++, but only C++ makes it easy and natural to use, and, moreover, C++'s is probably the most efficient.

In fact, MPW C++ provides a complex.h library (based on AT&T's complex number library) along with the standard libraries. I will use this library to illustrate how specific C++ features make it easy to use. In C, a library designer is restricted to defining structures and functions that operate on them. On the other hand, it is possible for a C++ library designer to create the appearance that the type he is defining is built into the language—C++ types can be “black boxes” with hidden members, the built-in operators and functions can be overloaded to operate on the new types, and automatic coercions can be specified between types. Furthermore, all of these can be done efficiently, without unnecessary overheads.

When reading this section, please refer to appendix A for one possible declaration of the C and C++ complex.h libraries.

Example

After including the complex.h library, to define a C++ procedure that returns the two roots of a quadratic equation, one would write:

```
// Solve a*x*x+b*x+c==0.
// root1 and root2 are the two roots.
void solveQuadratic(complex a, complex b,
  complex c, complex &root1, complex &root2)
 {
 complex d=sqrt(sqr(b)-4*a*c);
 root1=(d-b)/(2*a);
 root2=(d+b)/(-2*a);
 }
```

To solve $x^2+(7+3i)x+17=0$, one would use:

```
solveQuadratic(1,complex(7,3),17,root1,root2)
```

By comparison, to accomplish the same thing in C[1], one would need:

```
/* Solve a*x*x+b*x+c==0. */
/* root1 and root2 are the two roots. */
void solveQuadratic(complex a, complex b, complex
c, complex *root1, complex *root2)
 {
 complex d,constant;

 constant.re=4;
  /* Notice broken data abstraction here; */
 constant.im=0;
  /* constant's fields have to be accessed
     directly. */
 d=csqrt(csub(cmul(b,b),cmul(cmul(constant
   ,a),c)));
 *root1=cdiv(csub(d,b),cadd(a,a));
 constant.re=-2;
 *root2=cdiv(cadd(d,b),cmul(constant,a));
 }
```

To solve $x^2+(7+3i)x+17=0$ in C, one would use:

```
complex a,b,c;
a.re=1;
a.im=0;
b.re=7;
b.im=3;
c.re=17;
c.im=0;
solveQuadratic(a,b,c,&root1,&root2);
```

The C++ program is more readable and easier to modify, in addition to being more efficient. The C++ features that bring this about include good support for data abstractions, type coercion primitives, overloaded functions and operators, and inline functions. These features are described below.

Data Abstraction

An important difference between the two programs is that the C++ programmer need not be aware of the internal structure of a `complex` object. It happens to be represented as two `extended` numbers, but the user is not aware of this; in fact, the representation could be changed to use polar coordinates or a hybrid using rectangular and/or polar coordinates without affecting the user program[2]. This is the essence of data abstraction.

On the other hand, the C programmer is aware of the representation of a `complex` object—a structure containing the `re` and `im extended` fields—and, in fact, he has to manipulate it directly to initialize a `complex` variable to a real constant. This hinders any future changes to the structure of `complex` objects. One might argue that the library ought to provide a C function that initializes a `complex` variable to a real constant; unfortunately, such a function would impose excessive calling overhead for such a simple operation. A `#define` macro to initialize a `complex` variable would be more efficient, but `#define` macros have numerous scoping, aliasing, and redundant evaluation problems of their own.

Type Coercion

The C language performs some automatic type coercions on built-in types. For example, a `single` can be passed to a function that expects an `extended` argument, and a `char` can be used in an expression as an `int`. However, there are no automatic means to convert between user-defined types or between user-defined types and built-in types.

On the other hand, C++ lets the designer of a new type specify coercions to and from that type. This capability is valuable in many instances, as it lets user-defined types behave much like built-in ones. The complex.h library defines a coercion from an `extended` to a `complex` value by providing a constructor (described later) that constructs a `complex` number from an `extended` floating point number. The constructor performs the obvious mapping—the real number is converted to a complex number with the given real part and zero imaginary part.

The coercion is applied automatically whenever appropriate[3]. Thus, an `extended` value can be assigned to a `complex` variable; the coercion constructor will be called automatically. An `int` can also be stored in a `complex` variable; it will first be converted to `extended` and then to `complex`[4]. An integer can also be passed to a function that expects a `complex` argument, as is seen in the example above. The C++ version of `solveQuadratic` is called

[1] Also using the complex.h library supplied with MPW.

[2] I am ignoring here the roundoff and overflow issues that appear in some numerical programs. It is in general difficult to modify numerical algorithms without having some effect on rounding errors.

[3] There are some restrictions about ambiguous or multiple coercions, but they do not cause problems in practice.

[4] As will be seen later, many of these coercions are only conceptual and done by the compiler; there is usually no run-time penalty for coercing constants and for inlined coercions.

with the arguments `1` and `17`, which are coerced to type `complex` before the function is called. This makes the pro

gram much more readable than the C version, without sacrificing efficiency.

In addition to the above real-to-complex constructor, the complex.h library also declares a `complex` object constructor that takes two `extended` numbers to specify the real and imaginary parts. This constructor is seen in use as the second argument for `solveQuadratic`: `complex(7,3)`. Coercions which take more than one argument must be specified explicitly.

Overloading

Another indispensable feature of C++ in defining types such as `complex` is the ability to overload operators and function names. The complex.h library specifies functions which define the behaviors of such common C operators as `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, and `/=` when at least one of their arguments is `complex`. In addition, the library *overloads* a number of functions such as `sqrt`, `sin`, `cos`, `atan`, and `exp` by defining the behavior of these functions with `complex` arguments. Of course, the standard definitions of these overloaded operators and functions are still available when their arguments are not `complex`.

It is true that anything that can be done with overloaded functions can also be done without them, simply by picking unique names for all functions. However, this is often cumbersome, as the programmer has to remember the names of overloaded functions for all possible combinations of arguments. For example, the complex.h library specifies the behavior of `*` when both arguments are `complex`, when the first argument is `complex` and the second `extended`, and when the first argument is `extended` and the second `complex`[5] (in addition, of course, to C++'s built-in behavior of `*` when both arguments are `extended`). Three C functions with needlessly different names would have to be used to provide the same functionality, whereas in C++ all the programmer has to remember is that the usual multiplication operator `*` can now be used to multiply complex numbers.

Efficiency

With some work, the type coercion and function and operator overloading difficulties could be circumvented in C, at the expense of much less readable programs (compare the C and C++ versions of the example above). However, one more advantage of C is that many of the simple coercions and operations on `complex` values can be done without the overhead of a function call. C++ functions and constructors can be specified *inline*, requesting that that their code be duplicated wherever they are called instead of using a function call. For simple operations such as constructing a `complex` number out of an `extended` one or for adding two `complex` numbers, this inlining facility can result in significant time savings. Furthermore, an optimizing compiler can often perform a much better optimization job on an inlined function than on one that is compiled out-of-line.

Summary

Together, the data abstraction, type coercion, overloading, and inlining facilities of C++ combine to let a designer seamlessly extend the language by adding types to it that are indistinguishable from built-in ones. A reader of the C++ example above who is not familiar with C and C++ might assume that complex numbers are built into C++. In fact, with inline functions and a good optimizing compiler, an analysis of the code produced would support this illusion, as most `complex` value operations would be done inline. Thus, the above four areas of C++ combine to give programmers the power to extend the language—a power not present to the same degree in C, Pascal, or Object Pascal.

Other Examples

Complex numbers are not the only example of a useful language extension that can be seamlessly integrated into C++. [6] describes a library defining *associative arrays*—arrays that can be indexed by objects of non-cardinal types. For example, one can declare an array `a` indexed by strings, and then refer to `a["this"]` and `a["whatever string you like"]`. Interestingly enough, a straightforward reimplementation of the UNIX topological sort program `tsort` using associative arrays reduced that program to a trivial one-procedure program that actually

[5] Strictly speaking, only the first form (`complex*complex`) is necessary, as `extended` values will be automatically coerced to `complex` ones; however, the other two forms (`complex*extended` and `extended*complex`) are provided for efficiency.

ran faster than the conventional `tsort` because of the better algorithms used[6].

3. Constructors and Destructors

Unlike `complex` values in the previous section, many useful objects require initialization and destruction operations before and after they are used. Nevertheless, neither (Object) Pascal nor C provides any facilities for initializing and destroying user-defined objects. Since many complicated objects do need initialization and deallocation, programmers have defined conventions for doing these tasks. For example, many MacApp classes provide nontrivial initialization and destruction methods for their objects. Whenever a procedure creates a MacApp object of such a class, it must subsequently call the initialization method before it does anything else with the object. Similarly, to make sure that storage is properly deallocated, the procedure should call the `Free` method on every object that is to be destroyed.

Software conventions such as the one above are fine for simple tasks, but they do lead to errors and become unmanageable for complicated programs and object structures. Is it really feasible to ensure that `Free` is called on every object that should be freed, even if procedures return prematurely because of errors? My experience with this problem has been that unless a language provides support for automatic deallocation of objects, such deallocation bugs will remain in programs long after they have been written, debugged, and tested, especially if the error conditions that trigger the bugs are infrequent.

C++ offers a better solution to this problem through the use *constructors* and *destructors*. A constructor is a procedure associated with a class that is called whenever an instance object of that class is created. A destructor is a procedure that is called whenever any such instance object is destroyed. Constructors were mentioned briefly in the previous section, where they were used for type coercion. Constructors and destructors are not available in (Object) Pascal or C, although Pascal does use constructors internally to make sure that local `FILE` variables are properly initialized.

In C++, constructors and destructors are called regardless of whether their instance objects are local variables (stored on the stack) or dynamic heap objects (stored on the heap).

The `FileRec` Class

The listing below contains a sketch of an implementation of a simple `FileRec` class designed to keep track of a file to which data is being appended. A modified version of this class is used in TMON 3.0 to support printing.

```
// Class declaration
class FileRec {
private:
 HParamBlockRec pRec;
 bool isOpen;

public:
 FileRec();
 ~FileRec();

 OSErr open(short vRefNum, String fileName);
 OSErr close();

 OSErr write(char *);
 };



//Class implementation

//Constructor
FileRec::FileRec()
 {
 isOpen=0;
 }


//Destructor
FileRec::~FileRec()
 {
 close();
 }


OSErr FileRec::open(short vRefNum,
                    String fileName)
 {
 OSErr err;

 if (isOpen) return errIsAlreadyOpen;
 if (err=createFile('TEXT','MPS ',
                    vRefNum,fileName))
   return err;
 setupFileParamBlock(&pRec,vRefNum,
                     fileName);
 pRec.ioParam.ioPermssn=fsWrPerm;
```

[6]Of course, the same algorithms could be applied to improve the performance of the conventional C `tsort` program, but this would require more programmer effort.

```
 pRec.ioParam.ioMisc=0;
 if (err=PBHOpen(&pRec,0)) return err;
 isOpen=1;
 pRec.ioParam.ioPosMode=fsFromLEOF;
 pRec.ioParam.ioPosOffset=0;
 return PBSetFPos((ParamBlockRec *)&pRec,
                  0);
 }


OSErr FileRec::close()
 {
 OSErr err,err2;

 if (!isOpen) return 0;
 err=PBClose((ParamBlockRec *)&pRec,0);
 isOpen=0;
 err2=PBFlushVol((ParamBlockRec *)&pRec,
                 0);
 if (!err) err=err2;
 return err;
 }


OSErr FileRec::write(char *line)
 {
 if (!isOpen) return errIsNotOpen;
 pRec.ioParam.ioBuffer=line;
 pRec.ioParam.ioReqCount=strlen(line);
 pRec.ioParam.ioPosMode=fsAtMark;
 return PBWrite((ParamBlockRec *)&pRec,0);
 }
```

The function below shows an example of the use of a `FileRec`.

```
OSErr fileRecUser(short vRefNum)
 {
 OSErr err;
 FileRec file1;
 char str[256];
 int i;

 err=file1.open("MyFile",vRefNum);
 if (err) return err;
 err=file1.write("A disassembly of ROM "
                 "follows\n");
 if (err) return err;
 for (i=0x40800000; i<0x4087FFFF;
      i+=instLength(i))
   {
   err=disassemble(i,str);
   if (err) return err;
   err=file1.write(str);
   if (err) return err;
   if (userInterrupt()) return -1;
   }
 err=file1.close();
 printf("Done!\n");
 return err;
 }
```

The `fileRecUser` function first opens a file called “MyFile”, and then writes a disassembly of a ROM into it. Any errors are immediately reported.

The `FileRec` constructor is called on the `file1` local variable before the `fileRecUser` function begins executing, and simply sets the `isOpen` variable to false, marking `file1` as closed. The `FileRec` destructor is called when `fileRecUser` finishes. That destructor calls the `close` method, which checks whether the file is still open and, if so, closes it. Thus, `file1` is closed when the `fileRecUser` function returns, regardless of whether `fileRecUser` returns successfully at the end or via one of the internal `return` statements that caught an error.

The design of the `FileRec` class ensures two invariants:

- An open `FileRec` is always closed whenever it is deallocated.

- A `FileRec` file is closed only if it was open before, and it is closed exactly once.

Both of these conditions are crucial, and having the `FileRec` class enforce them improves a program's robustness. If the first condition were violated, a file might remain open for too long, which would prevent it from being opened again and might cause loss of data because the last block of data might not be flushed to the disk[7]. The consequences of violating

[7]Several commercial Macintosh programs have bugs that cause them to forget to close files when errors occur.

the second condition could be even more serious—it has been documented that closing a file twice or closing a file without successfully opening it first can erase a disk's directory! Again, the design of the `FileRec` class ensures that this catastrophic error will never happen; the user of the `FileRec` class does not have to worry about either of these problems and can, in fact, call the `close` method many times or not at all without adverse consequences.

Summary

Judicious use of constructors and destructors can greatly reduce the number of initialization and deallocation errors in a program. All objects can come to life initialized, and they can be automatically deallocated when they are no longer needed. Since initialization and deallocation errors are both common and elusive in C and Pascal, the use of C++ can significantly reduce the testing and debugging time for a program, in addition to freeing the programmer from the

burden of worrying about low-level programming issues.

4. Automatic Garbage Collection

The previous section introduced the use of constructors and destructors to initialize and clean up objects. This section will continue the same theme, but with a twist: I will demonstrate how it is possible to efficiently allocate and deallocate objects that can be *copied*. The `FileRec` objects from the previous section could not be copied because the two copies would still refer to the same file on the disk, which does not make much sense. Nevertheless, the copy operation is essential for many primitive data types such as strings, infinite precision numbers, trees, matrices, expressions, and others. In this section I will describe a simple `String` class that permits "transparent" copying.

The `String` Class

One of the more annoying features of many C programs is their tendency to impose arbitrary limits on string lengths. Names, input lines, *etc.* are often limited in size, and exceeding the limits causes an error message at best and anything from a system crash to memory corruption at worst. A large length limit might waste memory in most applications yet still be too small for some purposes. The infamous Internet worm [8] propagated across the country in part by exceeding the maximum length of a string.

TMON 3.0 must be as reliable and flexible as possible, and I could not impose any arbitrary limit on the length of a string. Most strings are small, but it is possible that someone might want to examine a string that is a million bytes long; TMON 3.0 must handle such a request, as long as it has enough memory to do so (and if it doesn't, it must say so and abort the operation gracefully). Moreover, allowing arbitrary-length strings led to a simplification of TMON's code—entire files can now be represented as strings, and printing a window amounts to mostly concatenating data to a string. A View window in TMON 3.0 is little more than a string viewer.

One obvious way to store arbitrary-length strings is as handles on a heap. Unfortunately, manipulating handles directly is error-prone and tedious. To remedy this situation, I defined a special C++ `String` class to make `String` manipulation even easier than it is with fixed-length strings in C.

Operations

The operations defined for `String`s include copying, conversion to and from "standard" C-style strings, measuring their lengths, extracting characters, concatenating, and concatenating in place. Copying is invoked using the `=` assignment operator. The `[` and `]` brackets are used to extract characters from `String`s, just as for "standard" C-style strings. Concatenation is represented using the `|` operator, and concatenation in place using the `|=` and `^=` operators.

Examples

Suppose that `a` and `b` are `String`s containing `"Alyssa"` and `"Ben"`, respectively.

`length(a)` will return `6`.

`a[2]` will return the character `'y'`. The `[]` operation is bounds-checked, so `a[6]`, `a[-17]`, and `a[999]` will all return the null character `'\0'`.

`a|b` will return a new `String` containing `"AllyssaBen"` without altering `a` or `b`. If there is not enough memory to allocate a new `String`, `a|b` will return the null string; the `String` class could be modified to do something else in this situation.

`String c="Cindy"` creates a new `String` variable and initializes it to `"Cindy"`.

`String d=a|", "|b|", and "|c` creates a new `String` variable and initializes it to `"Alyssa, Ben, and Cindy"`. Notice how the standard C-style strings can be mixed with `String`s in expressions.

`a=b` destroys the previous `String` in `a` and assigns `"Ben"` to `a`. The assignment does not involve copying actual string contents—both `a` and `b` now point to the same `StringData` record (defined later).

`a|=c` concatenates `"Cindy"` to `a`, thereby placing `"BenCindy"` in `a`. `b` is not altered—it still contains `"Ben"`.

`a.substr(2,4)` returns `"nCin"`—a new `String` containing four characters from `a` starting at offset 2.

`a.cString()` returns a pointer to a C string (i.e. `char *`) containing the characters `"BenCindy"` in `a`. The C string is located in an unlocked handle, so it should be copied before any routine that can move the heap is called.

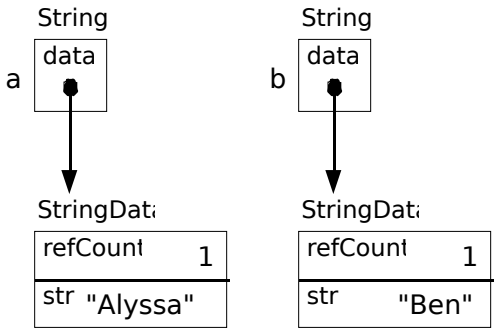
Copying `String`s

The previous section demonstrated how it is possible to arrange for a `String` variable to be initialized when it is created and deallocated when it is destroyed. However, the fact that `String`s can be copied leads to complications. The simple approach to dealing with the assignment `a=b` where `a` and `b` are `String`s would be to copy `String b`'s data to `String a`. I rejected this approach because it is both slow and unnecessarily wastes memory. Instead, I chose to keep a reference count in each `String`'s data that indicates how many `String` variables refer to it. When assigning `String b` to `a` in `a=b`, the reference count of `String b` is incremented by one (because `String a` now points to it too), and `String a`'s reference count is decremented by one. A `String`'s data is deallocated if and only if its reference count reaches zero, for then there are no more references to it.

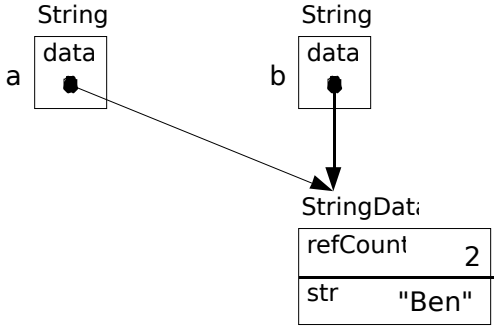
A `String` is just a handle to a structure containing the reference count and the actual string data. It is declared as

```
struct StringData
 {
 int refCount;
 char str[1]; //Variable length
 };

class String
 {
private:
 StringData **data;

 //operations go here.
 };
```

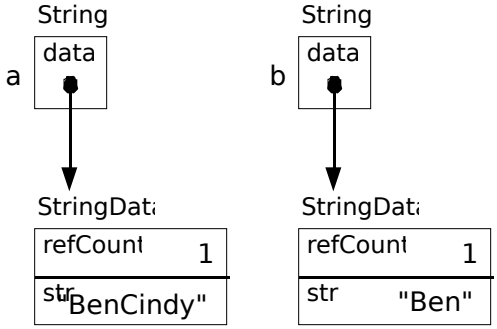
At the beginning of the series of examples above, the `String` variables `a` and `b` were in the following state[8]:



After the assignment `a=b`, the variables changed to:



After `a|=c`, the variables changed to:



Altering `String`s

The `String` class defines the operations `^=` and `|=` that allow `String`s to be modified. These operators modify the `StringData` in-place whenever possible. However, since many `String` variables can point to the same data, the data may have to be copied before it is altered because any modifications should only be visible through the one `String` variable to which the modification is applied. Thus, when `a|=c` was done in the examples above, `a`'s `StringData` had to be copied because `b` was also pointing to it. The copy would not have been necessary had no other variable pointed to `a`'s `StringData`.

[8]The arrows in these figures represent handles, not pointers.

Garbage Collection

The idea of using reference counts to decide when objects should be deallocated is known as *reference-counting garbage collection*. This scheme is *incremental*, in that it does not exhibit the long pauses encountered with some other forms of garbage collection. Moreover, reference-counting garbage collection is guaranteed to deallocate objects as soon as is possible as long as no cyclical structures are allocated. Since `String`s do not contain any pointers in their data, this condition is trivially satisfied.

In order to correctly implement garbage collection, the `String` class designer must be able to override C++'s default behavior on assignment and copying of `String` objects in order to properly adjust the reference counts. Fortunately, this is easy in C++, whereas it is impossible in C or Pascal.

C++ objects can be copied in the following situations:

1. Initialization of a new local or heap variable using an existing value (i.e. `String a=b` for initializing the local variable `a` or `String *a=new String(b)` for allocating a `String` variable on the heap and storing a pointer to it in `a`).

2. Assignment of an expression to a variable (i.e. `a=b` or `myStruct->a=b`).

3. Passing a value as a parameter to a function (i.e. `length(a)`).

4. Returning a value from a function.

Furthermore, the above situations also apply if a structure including `String` fields is initialized, assigned, passed to a function, or returned[9].

In situations 1, 3, and 4, a `String` is stored into a new variable that has not been initialized, so C++ calls a special `String` constructor, which takes another `String` as an argument. By providing this constructor, it is possible to increment the `String`'s reference count appropriately.

In situation 2, a `String` is stored into a variable that already contains a `String`, so the reference count of the old `String` must be decremented before the new `String` is stored in the destination variable and its reference count incremented. The `String` class does this by overriding the assignment operator `operator=` for objects of type `String`.

Finally, the `String` class takes care of simple `String` creation and deallocation using constructors and destructors as described in the previous section.

Implementation

The implementation of the sample `String` class is given in appendices B and C. That `String` class is a greatly simplified version of one of the main string classes in TMON 3.0. The full class in TMON 3.0 is actually derived from a series of storage-managing ancestor classes. For simplicity I removed automatic locking, purging, and nontrivial character manipulation and speed enhancement features from the `String` class presented here. Nevertheless, even the simple `String` class is useful for practical applications.

Example

In appendix D, I present a sample MPW tool, StringSample, that uses the `String` class to parse a Macintosh file pathname into its components. There is no restriction on the length of the original pathname, other than available memory. The pathname is separated into a volume name, a compound directory name, and the leaf file name. In addition, StringSample proceeds to strip directories off the compound directory name, giving the complete list of directories that form the pathname. StringSample is purely a demonstration of string manipulations; it makes no calls to the file system, although it could be used as a component of a class that does.

Sample output of the StringSample tool is below. StringSample parses all of its command line arguments into components. For the last argument, StringSample lists the access paths to all of the intermediate directories on the way to the specified file.

StringSample "*LocalFile*" ∂
  "*MyVolume:MyFile*" ∂
  "*:LocalDir:Dir2:Dir3::File:*"

```
"LocalFile":
 VolumeName=""
 DirName=":"; FileName="LocalFile"
```

[9]One major bug in pre-2.0 C++ was that the compiler did not recognize this possibility, which made it difficult to use `String`s or any other sophisticated objects as fields in structures.

```
"MyVolume:MyFile":
```

```
 VolumeName="MyVolume"
 DirName=":"; FileName="MyFile"

":LocalDir:Dir2:Dir3::File:":
 VolumeName=""
 DirName=":LocalDir:Dir2:Dir3::";
         FileName="File"
 DirName=":LocalDir:Dir2:Dir3:";
         FileName=""
 DirName=":LocalDir:Dir2:";
         FileName="Dir3"
 DirName=":LocalDir:"; FileName="Dir2"
 DirName=":"; FileName="LocalDir"
```

Summary

In this section I presented the `String` class which implements low-overhead, robust, and easy-to-use operations on arbitrary-length strings. The ease of use and robustness contribute significantly to programmer productivity by virtually eliminating storage errors, while contributing to efficiency by not copying large strings unless necessary—strings are always allocated and deallocated as needed, nothing is deallocated early or twice, and there are no memory leaks.

Setting up a reference-counting garbage collection in (Object) Pascal or C is not feasible because there are too many opportunities to make reference-counting errors, especially during program maintenance. Such errors are often very hard to find and may remain dormant for years before striking.

5. Beyond Individual Types

In this section I will take the automatic garbage collection theme one step further and sketch how it can be used to leverage the development of a program.

Inheritance

Inheritance is the ability to define a new object based on an existing one by describing the differences. For example, a `Window` is basically a `GrafPort` that permits additional operations; the `Window` is said to *inherit* from a `GrafPort`, and the `GrafPort` is `Window`'s superclass.

Whereas inheritance had to be faked using type coercion in Pascal and C, it is a natural capability of C++ and, in fact, most object-oriented languages. I will not write about the benefits of inheritance here; many tutorials have been written on inheritance, and one need only look at MacApp for examples of effective uses of inheritance.

Nevertheless, there is one aspect of C++'s inheritance that is often not appreciated: C++ allows the fields of a class's object to contain values which are themselves objects of other classes. This kind of type inclusion is not common in object-oriented languages and leads to improved efficiency by reducing the number of storage management calls needed. Furthermore, operations on object fields are in some restricted ways inherited in the operations on the object itself. The example below should clarify this confusing concept.

Suppose the class `PathName` were defined as follows:

```
struct PathName
 {
 short vRefNum;
 String volName;
 int dirID;
 String dirName;
 String fileName;
 };
```

With this definition, unless they are overridden, `PathName` will have predefined constructors, destructors, and assignment operators. The default `PathName` constructor will initialize all three `String`s in the `PathName` to null values. The default `PathName` destructor will deallocate all three `String`s. Finally, assigning one `PathName` object to another will call the `String` assignment operator on all three `String`s, allowing the reference counts to be adjusted properly. The benefits of C++ show up once again here, in that types using garbage-collected fields automatically inherit the benefits of garbage collection.

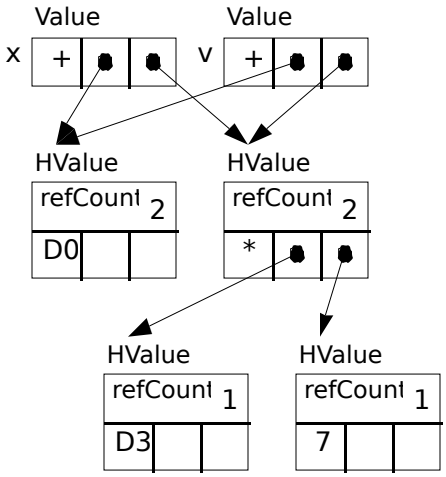
The Value Class

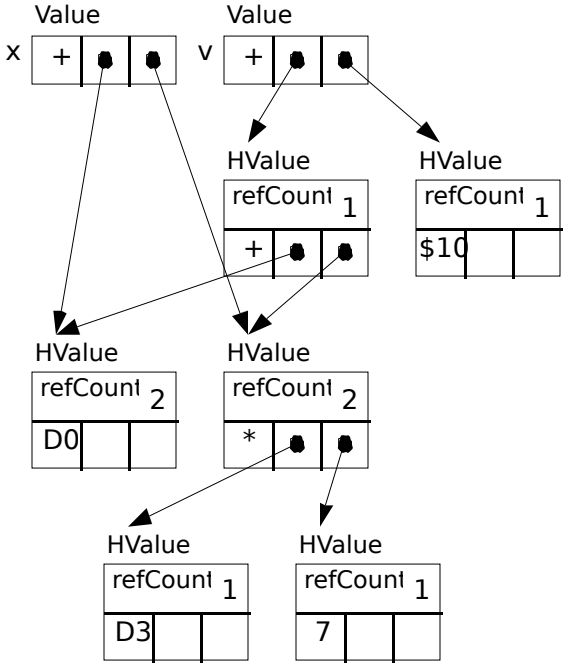
The two forms of inheritance mentioned in the previous subsection permit simple development of large collections of classes, all inheriting from a small garbage-collection kernel. The TMON 3.0 `Value` class is one of the best examples of the power of this approach. A TMON 3.0 `Value` is anything that can be stored in a TMON variable—an integer, a floating point number, a `String`, an unlimited-length block of bytes, a TMON error code, a TMON type, or a delayed expression consisting of a function applied to

other `Value`s. The last variant is the most interesting—the function can be any of TMON's arithmetic operators or functions, while the `Value`s can themselves be delayed expressions, allowing delayed expressions to be built of arbitrary complexity. The declaration of a TMON `Value` is sketched below[10].

```
enum ValKind
{anError,aDelay,anInteger,aFloat,aDatum,
aString,aType};

class Value
 {
private:
 ValKind kind;
 short length;
 union {
   OSErr errorVal;
   int intVal;
   Type *typeVal;
   float fSingle;
   double fDouble;
   extended fExtended;
   extended10 fExtended10;
   extended12 fExtended12;
   String stringVal;
   Block blockVal;
   struct {
     Function f;
     class HValue **args[maxArgs];
     } delayedVal;
   };

public:
 //Constructors
 Value();
 Value(const Value &);
 Value(int);
 Value(extended);
 Value(const String &);

 //Destructor
 ~Value();

 //Assignment
 Value &operator=(const Value &);

 //Operations
 bool operator==(const Value &) const;

 void neg();
 void logNot();
 //...
 void operator+=(const Value &arg1);
 void operator-=(const Value &arg1);
 //...

 //Evaluate delayed Value.
 Value force() const;
 //Compile delayed Value to 68000 code.
 CompiledValue compile() const;
 //Print Value to standard output.
 void print() const;
 };

class HValue:Value
 {
 int refCount;
 };
```

Discussion





[10]The declaration actually used in TMON 3.0 is slightly different due to a few pesky "features" of the C++ language.

Note the flexibility of the design of the `Value` class. Suppose that `v` and `w` are `Value`s. To add `w` to `v`, one would simply write `v+=w`. To add the number `$10` to `v`, one would use `v+=$10`; the

`$10` is automatically coerced to an integer `Value`. If `v` happened to contain the integer `$40800000` before this addition, it would now contain `$40800010`. On the other hand, if `v` and `x` both contained the delayed expression `ΔD0+D3*7`, `v` would now contain `ΔD0+D3*7+$10`, as shown in the figure above.

TMON functions need not handle delayed expressions specially. When a function returns, the `Value` destructor is called to automatically update the reference counts and deallocate storage of `Value`s that were stored in its local variables. The same thing happens when a structure containing `Value`s is disposed. When a complicated `Value` is deallocated, it recursively deallocates its components. Copying `Value`s and passing them as parameters are both efficient—a small structure is copied and one reference count updated.

Summary

The C++ inheritance capabilities permit one to write a few garbage-collecting classes and then use them to define entire hierarchies of objects that automatically initialize, keep track, and dispose of themselves. Thus, the safety-critical garbage collection logic can be limited to a small part of the program. Other classes can be easily designed without introducing handle dereferencing or deallocation problems. This combination of features saved a large amount of time in the development of TMON 3.0. Without these capabilities, even after months of debugging, I would not have found all of the errors in TMON's `Value` management; moreover, making modifications to TMON would have become virtually impossible due to the constant peril of missing a garbage collection reference or deallocating something twice.

My experiences in this area are not unique. Stephen Wolfram, in writing his symbolic manipulation program *Mathematica* [13], found it easier to implement and use a new language built on top of C than to try to write the program directly and struggle to find all of the storage allocation bugs. Fortunately, nowadays it is possible to achieve the same goal by using C++ instead of implementing an entirely new language.

6. Conclusion

The Good News

In this paper I presented four C++ capabilities that allow programmers to design sophisticated, self-contained types: convenient and efficient type operation syntax, type constructors and destructors, garbage collection, and inheritance. With these capabilities it is now possible to design efficient and robust types that virtually eliminate entire classes of program bugs such as storage deallocation or handle dereferencing errors. In addition to saving considerable debugging time, these capabilities free programmers to concentrate on higher-level considerations in their programs and make it much easier to modularize programs. It is also possible to use the concepts described here to write efficient, reusable modules of code.

The Bad News

Unfortunately, the productivity gains described in this paper do come at a price. To realize these gains, one must learn the advanced capabilities of C++, and one must commit to using C++ throughout the entire program. It is difficult to mix “advanced” C++ code with code written in other languages, although it is possible to do so if one is extremely careful.

Commitment

The use of C++'s advanced features in a project implies a commitment to use C++ throughout the project. It is not possible to mix C++ with other languages such as (Object) Pascal, C, or assembly at a high level while still realizing the benefits of the capabilities described in this paper. Of course, one can mix C++ with (Object) Pascal, C, or assembly at a low level, but one cannot pass objects with constructors, destructors, and virtual functions across language boundaries without exercising considerable care[11]. Thus, although TMON 3.0 is written half in C++ and half in assembly language, the assembly language half is almost exclusively concerned with low-level functioning of TMON. Objects such as `String`s and `Value`s are almost never passed to assembly language routines.

[11]MPW C++ does contain a compatibility hack to let it share virtual functions with Object Pascal; unfortunately, some of C++'s advantages are lost when this is done.

For this reason, any future user areas and extensions to TMON must also be written in C++.

Potential for Abuse

It is easy to abuse C++. C++ will let one redefine the comma operator to perform addition, use * to mean "print", and give the same name to three completely unrelated functions; all of these can be extremely useful in utterly confounding anyone who subsequently tries to read or modify the program. Moreover, in group projects it is easy for several people to simultaneously develop their own hierarchies of utility classes, producing needlessly large and complicated programs[12].

C++ works best for programmers who are familiar with the concepts of abstraction (information hiding). In team projects involving programmers not familiar with abstraction, I found that problems arose with abstractions being broken, resulting in eventual bugs and unmaintainability of the program[13]. On the other hand, team projects in which abstractions were followed went smoothly. C++ helps in this regard by providing for hiding instance variables, but these provisions will not deter a programmer working against a schedule from breaking an abstraction.

Future Evolution

C++ is not yet a complete language, and some capabilities are sorely needed. The most urgently needed facilities are type polymorphism, exception handling, and support for fine-grain concurrency, locking, and simple atomic transactions[14]. Many modern programming languages, including Ada [1], Common Lisp [9], and Modula-3 [2] provide good support for at least the first two; yet, even C++ is still in the stone age by providing no support in these areas. In the development of TMON I was forced to use *ad hoc* mechanisms to implement these capabilities—I used return codes to simulate exception handling and extensive type coercions to simulate polymorphic types. Neither solution is satisfactory, though, and implementing these kludges caused numerous bugs in addition to making TMON hard to modify. Fortunately, Bjarne Stroustrup is working on correcting these difficulties, and we can

[12]One solution sometimes adopted to solve this problem is to appoint an "object czar," but this solution, if done improperly, will introduce more bureaucracy than it is worth.

[13]Nevertheless, my problems are minor compared with Apple's difficulties with developers sneaking behind the system software abstraction barrier. However, to be fair, some of the blame belongs on Apple for either not providing sufficient functionality in the system software interface for developers to do their jobs or for leaving bugs in the operating system requiring breaking the system software abstraction to work around them.

[14]Some of the latter three could be provided by libraries, but better definitions of the language's semantics are needed for portability; otherwise, concurrent programs will not be portable from one C++ implementation to another.

look forward to new and improved releases of C++ in the future.

Summary

C++ is like a power tool, where C and Object Pascal are like hand tools. To someone experienced with C++, it can be a tremendous time-saver. However, someone just starting to use a power tool such as C++ should be careful. It is easy to forget to use the proper guards and injure oneself until one learns the safety rules. When starting with C++, I recommend following a C++ safety manual such as Apple's style guide [3], also found in an appendix of the MPW C++ manual [7].

A. Sample `complex` Library Declarations

Portions of a possible declaration of the C and C++ complex.h libraries are sketched below. See the MPW complex.h interface file for the full version.

C:

```
struct complex {
 extended re,im;
 };

typedef struct complex complex;

//Arithmetic operations
complex cadd(complex, complex);
complex csub(complex, complex);
complex cmul(complex, complex);
complex cdiv(complex, complex);

//Functions
complex csqrt(complex);
...
```

C++:

```
struct complex {
private:
 //Private variables and functions cannot be accessed outside the complex class.
 extended re,im;

public:
 //Constructors
 //Since the definitions are given here, the code will be automatically inlined.
 complex()
   {}
 complex(extended r)
   {re=r; im=0.0;}
 complex(extended r, extended i)
   {re=r; im=i;}

 //Operators
 complex operator+(complex);
 complex operator-(complex);
 complex operator*(complex);
 complex operator/(complex);
 ...

 //Functions
 friend complex sqrt(complex);
 ...
 };

//Inline definitions of simple operators and functions must be present in the declaration.
inline complex::operator+(complex z)
 {
 return complex(re+z.re,im+z.im);
 }
...
```

B. `String` Class Declaration

This appendix contains the declaration of the `String` class and library described in section 4. The definition of this class is given in the next appendix.

HString.h:

```
// Copyright 1990 Waldemar Horwat.
// Permission is granted for noncommercial use of this code.

typedef int bool;

struct StringData:HandleObject
 {
 int refCount;
 char str[1]; //Variable length
 };

class String
 {
private:
 StringData *data; // May be NIL!

 void internalCopy(const String &src);
 OSErr internalCopy(const char *cString);
 OSErr fresh();

public:
 // Cumulative error.
 static OSErr err;

 // Constructors
 String() {data=0;}
 String(const String &src) {internalCopy(src);}
 String(const char *cString) {internalCopy(cString);}

 // Destructor
 ~String();

 // Clear the String to a null string.
 void clear();

 // Assignment operators
 String &operator=(const String &src);
 OSErr operator=(const char *cString);

 // Query functions.
 int length() const;
 char operator[](int index) const;
 String substr(int offset, int length) const;
 //Result may move when heap is scrambled!
 char *cString() const;

 bool operator==(const String &src2) const;

 // Concatenation
 String operator|(const String &src) const;

 // Appending data to the end
 OSErr operator|=(const String &src);
 OSErr operator|=(char ch);
```

```
 OSErr operator|=(char *cString);
 // Prepending data to the beginning
 OSErr operator^=(char ch);
 OSErr operator^=(char *cString);
 };
```

C. `String` Class Definition

This appendix contains the definition of the `String` class and library described in section 4. The declaration of this class is given in the previous appendix.

HString.c:

```
// Copyright 1990 Waldemar Horwat.
// Permission is granted for noncommercial use of this code.

#include <Types.h>
#include <Memory.h>
#include <OSUtils.h>
#include <String.h>
#include <Errors.h>
#include "HString.h"


OSErr String::err=0;

/*————————————————————————————————————————————————————————————————————————————*/
/* Copy src to this without deallocating whatever was in this previously.         */
/*————————————————————————————————————————————————————————————————————————————*/
void String::internalCopy(const String &src)
 {
 if (data=src.data) data->refCount++;
 }


/*————————————————————————————————————————————————————————————————————————————*/
/* Copy cString to this without deallocating whatever was in this previously.     */
/*————————————————————————————————————————————————————————————————————————————*/
OSErr String::internalCopy(const char *cString)
 {
 int length=strlen(cString);

 if (data=(StringData *)NewHandle(length+5))
   {
   data->refCount=1;
   BlockMove(Ptr(cString),data->str,length+1);
   return 0;
   }
 return String::err=memFullErr;
 }
```

```
/*-------------------------------------------------------------------------------------------*/
/* Make sure that this String's data is not shared with any other String.  This              */
/* function should be called before this String's data is altered.                           */
/*-------------------------------------------------------------------------------------------*/
OSErr String::fresh()
 {
 OSErr err;

 if (!data || data->refCount<=1) return 0;
 StringData *data2=data;
 if (err=HandToHand((Handle *)&data2)) return String::err=err;
 data->refCount--;
 data2->refCount=1;
 data=data2;
 return 0;
 }


/*-------------------------------------------------------------------------------------------*/
/* Deallocate this String.  Deallocate the data too if its reference count reaches           */
/* zero.                                                                                     */
/*-------------------------------------------------------------------------------------------*/
String::~String()
 {
 if (data && !(--data->refCount))
   DisposHandle(Handle(data));
 }


/*-------------------------------------------------------------------------------------------*/
/* Clear this String to a null string.                                                       */
/*-------------------------------------------------------------------------------------------*/
void String::clear()
 {
 this->String::~String();
 data=0;
 }


/*-------------------------------------------------------------------------------------------*/
/* Assign src to this.  Deallocate the old String in this.                                   */
/*-------------------------------------------------------------------------------------------*/
String &String::operator=(const String &src)
 {
 if (this!=&src) //Handle a String assigned to itself correctly!
   {
   this->String::~String();
   internalCopy(src);
   }
 return *this;
 }


/*-------------------------------------------------------------------------------------------*/
/* Assign the cString to this.  Deallocate the old String in this.                           */
/*-------------------------------------------------------------------------------------------*/
OSErr String::operator=(const char *cString)
 {
 this->String::~String();
 return internalCopy(cString);
 }
```

```
/*————————————————————————————————————————————————————————————————————————*/
/* Return the length of this String.                                          */
/*————————————————————————————————————————————————————————————————————————*/
int String::length() const
 {
 if (!data) return 0;
 return strlen(data->str);
 }

/*————————————————————————————————————————————————————————————————————————*/
/* Return the character at position index in this String.  Return the null character if */
/* index is out of bounds.                                                    */
/*————————————————————————————————————————————————————————————————————————*/
char String::operator[](int index) const
 {
 if (!data || index<0 || index>=strlen(data->str)) return 0;
 return data->str[index];
 }


/*————————————————————————————————————————————————————————————————————————*/
/* Return the substring of the given length starting from the given offset.  The null   */
/* string is returned if the offset is out of bounds.  The String returned may be       */
/* shorter than length characters if offset+length exceeds the length of this String.   */
/*————————————————————————————————————————————————————————————————————————*/
String String::substr(int offset, int length) const
 {
 String result; //Defaults to null.
 int thisLength=String::length();

 if (offset>=0 && offset<thisLength)
   {
   if (offset+length>=thisLength) length=thisLength-offset;
   if (result.data=(StringData *)NewHandle(length+5))
     {
     result.data->refCount=1;
     BlockMove(data->str+offset,result.data->str,length);
     result.data->str[length]='\0';
     }
   }
 return result;
 }


/*————————————————————————————————————————————————————————————————————————*/
/* Return the contents of this String as a C string.  The result may move the next time */
/* memory is allocated!                                                       */
/*————————————————————————————————————————————————————————————————————————*/
char *String::cString() const
 {
 if (!data) return "";
 return data->str;
 }


/*————————————————————————————————————————————————————————————————————————*/
/* Return true if the Strings are exactly equal.                              */
/*————————————————————————————————————————————————————————————————————————*/
bool String::operator==(const String &src2) const
 {
 if (!data || !src2.data)
   return length()==src2.length();
 return !strcmp(data->str,src2.data->str);
 }
```

```
/*————————————————————————————————————————————————————————————————————————————*/
/* Concatenate this and src to yield a new String.                                */
/*————————————————————————————————————————————————————————————————————————————*/
String String::operator|(const String &src) const
 {
 String result=*this;
 result|=src;
 return result;
 }


/*————————————————————————————————————————————————————————————————————————————*/
/* Destructively concatenate src to the end of this.                              */
/*————————————————————————————————————————————————————————————————————————————*/
OSErr String::operator|=(const String &src)
 {
 OSErr err;

 int srcLength=src.length();
 if (!srcLength) return 0; // Nothing to concatenate?
 if (!data)
   {
   internalCopy(src);
   return 0;
   }
 if (err=fresh()) return err;
 SetHandleSize(Handle(data),GetHandleSize(Handle(data))+srcLength);
 if (err=MemError()) return String::err=err;
 strcat(data->str,src.data->str);
 return 0;
 }


/*————————————————————————————————————————————————————————————————————————————*/
/* Destructively concatenate ch to the end of this.                               */
/*————————————————————————————————————————————————————————————————————————————*/
OSErr String::operator|=(char ch)
 {
 char cString[2];

 cString[0]=ch;
 cString[1]='\0';
 return *this|=cString;
 }


/*————————————————————————————————————————————————————————————————————————————*/
/* Destructively concatenate cString to the end of this.                          */
/*————————————————————————————————————————————————————————————————————————————*/
OSErr String::operator|=(char *cString)
 {
 OSErr err;

 int srcLength=strlen(cString);
 if (!srcLength) return 0; // Nothing to concatenate?
 if (!data)
   {
   internalCopy(cString);
   return 0;
   }
 if (err=fresh()) return err;
 SetHandleSize(Handle(data),GetHandleSize(Handle(data))+srcLength);
```

```
 if (err=MemError()) return String::err=err;
 strcat(data->str,cString);
 return 0;
 }


/*————————————————————————————————————————————————————————————————————————*/
/* Destructively concatenate ch to the beginning of this.                     */
/*————————————————————————————————————————————————————————————————————————*/
OSErr String::operator^=(char ch)
 {
 char cString[2];

 cString[0]=ch;
 cString[1]='\0';
 return *this^=cString;
 }

/*————————————————————————————————————————————————————————————————————————*/
/* Destructively concatenate cString to the beginning of this.                */
/*————————————————————————————————————————————————————————————————————————*/
OSErr String::operator^=(char *cString)
 {
 OSErr err;

 int srcLength=strlen(cString);
 if (!srcLength) return 0; // Nothing to concatenate?
 if (!data)
   {
   internalCopy(cString);
   return 0;
   }
 if (err=fresh()) return err;
 Size dataSize=GetHandleSize(Handle(data));
 SetHandleSize(Handle(data),dataSize+srcLength);
 if (err=MemError()) return String::err=err;
 BlockMove(data->str,data->str+srcLength,dataSize-4);
 BlockMove(cString,data->str,srcLength);
 return 0;
 }
```

D. StringSample Code

This appendix contains an MPW tool that illustrates the use of the `String` class. The tool is described in a bit more detail in section 4.

StringSample.make:

```
StringSample ƒ StringSample.c.o HString.c.o
 Link -c 'MPS ' -t MPST ∂
       StringSample.c.o HString.c.o ∂
       "{CLibraries}"CPlusLib.o ∂
       "{CLibraries}"StdCLib.o ∂
       "{CLibraries}"CInterface.o ∂
       "{CLibraries}"CRuntime.o ∂
       "{Libraries}"Interface.o ∂
       -o StringSample

HString.c.o ƒ HString.c HString.h
 CPlus HString.c
StringSample.c.o ƒ StringSample.c HString.h
 CPlus StringSample.c
```

StringSample.c:

```
// Copyright 1990 Waldemar Horwat.
// Permission is granted for noncommercial use of this code.

#include <Types.h>
#include <Memory.h>
#include <Errors.h>
#include <stdio.h>
#include "HString.h"

/*————————————————————————————————————————————————————————————————————————————*/
/* Split the path name in pathName into the volume name (without the colon) that is     */
/* stored in volumeName and everything following it, which is stored in                 */
/* partialPathName.  A colon is prepended to partialPathName if it didn't already have  */
/* one.  Both volumeName and partialPathName may be nil, in which case they are not     */
/* returned.  Both volumeName and partialPathName can alias with pathName.              */
/*————————————————————————————————————————————————————————————————————————————*/
OSErr splitVolumeName(const String &pathName, String *volumeName, String *partialPathName)
 {
 String::err=0;
 int length=pathName.length();

 for (int i=0; i<length && pathName[i]!=':'; i++);
 if (i==length)
   {
   if (volumeName) volumeName->clear();
   if (partialPathName)
     {
     *partialPathName=pathName;
     (*partialPathName)^=':';
     }
   }
 else
   {
   String pathName2=pathName;
   if (volumeName) *volumeName=pathName2.substr(0,i);
   if (partialPathName) *partialPathName=pathName2.substr(i,length-i);
   }
 return String::err;
 }


/*————————————————————————————————————————————————————————————————————————————*/
/* Split the path name in partialPathName into the last name (optionally followed by a  */
/* colon, which is removed) that is stored in fileName and everything preceding it,     */
/* which is stored in dirName.  Both fileName and dirName may be nil, in which case     */
/* they are not returned.  Both fileName and dirName can alias with partialPathName.    */
/*————————————————————————————————————————————————————————————————————————————*/
OSErr splitPartialPathName(const String &partialPathName, String *dirName, String *fileName)
 {
 String::err=0;
 OSErr err=0;
 String partialPathName2=partialPathName;
 int length=partialPathName2.length();

 if (length && partialPathName2[length-1]==':') length--;
 if (!length) err=bdNamErr;
 else
   {
   for (int i=length; i>0; --i)
     if (partialPathName2[i-1]==':') break;
   if (dirName) *dirName=partialPathName2.substr(0,i);
```

```
    if (fileName) *fileName=partialPathName2.substr(i,length-i);
    }
  if (!err) err=String::err;
  return err;
  }


OSErr parseFile(char *pathNameStr)
  {
  printf("\"%s\":\n",pathNameStr);

  String::err=0;
  String pathName=pathNameStr;
  OSErr err=String::err;
  String volumeName;
  String partialPathName;
  String fileName;

  if (!err)
    {
    err=splitVolumeName(pathName,&volumeName,&partialPathName);
    if (!err)
      {
      // I am not sure this is right; printf could allocate memory?
      printf("  VolumeName=\"%s\"\n",volumeName.cString());
      while (partialPathName.length())
        {
        err=splitPartialPathName(partialPathName,&partialPathName,&fileName);
        if (err)
          {
          if (err==bdNamErr) err=0;
          break;
          }
        printf("  DirName=\"%s\"; FileName=\"%s\"\n",
               partialPathName.cString(),fileName.cString());
        }
      }
    }
  return err;
  }


void main(int argc, char **argv)
  {
  int i;

  for (i=1; i<argc; i++)
    {
    OSErr err=parseFile(argv[i]);
    if (err) printf("(Error $%X)\n",err);
    printf("\n");
    }
  }
```

Bibliography

[1] Serafino Amoroso and Giorgio Ingargiola. *Ada: An Introduction to Program Design and Coding*. Howard W. Sams & Co., Marshfield, MA, 1985.

[2] Luca Cardelli, *et al*. *Modula-3 Report*. Olivetti Research Center/DEC report.

[3] David Goldsmith and Jack Palevich. "Unofficial C++ Style Guide." *Develop*, Issue 2, April 1990.

[4] Waldemar Horwat. "Parallel Processing Paradigms." MacHack '89.

[5] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*, Second Edition. Prentice Hall, 1988.

[6] Andrew Koenig. "Associative Arrays in C++." *Proceedings of Summer USENIX '88*.

[7] Macintosh Programmer's Workshop 3.1 C++ Reference Manual. Apple Computer, Inc., 1990.

[8] Donn Seeley. "A Tour of the Worm." *Proceedings of Winter USENIX '89*.

[9] Guy L. Steele. *Common Lisp: The Language*, Second Edition. Digital Press, Digital Equipment Corporation, 1990.

[10] Leon Sterling and Ehud Shapiro. *The Art of Prolog*. MIT Press, Cambridge, MA, 1986.

[11] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, 1987.

[12] Bjarne Stroustrup. "The Evolution of C++: 1985 to 1989." Included as documentation with AT&T and MPW C++ compilers.

[13] Stephen Wolfram. *Mathematica: A System for Doing Mathematics by Computer*. Addison-Wesley, 1988.