Reducing Run-Time Overhead in C++ Programs
Embedded Systems Conference San Francisco 2002
Classes 405 and 445

Dan Saks

Introduction

As a group, embedded systems programmers tend to be more concerned than other programmers about the speed and space costs of programming language features, and rightly so. These concerns lead some of them to reject certain features outright in the belief that those features are too costly. In the process, they may adopt poor designs and coding techniques just to avoid those language features. This is often a mistake, because C++ offers many ways to fine-tune programs to eliminate those costs, or at least reduce them to acceptable levels.

The Big Picture

With the possible exception of the smallest, most constrained applications, typical embedded applications are just some combination of capturing input, storing and retrieving data, performing computations, and generating output. In other words, most embedded programming is just plain programming, and therefore, by and large, good embedded programming technique is just plain good programming technique.

You should resist letting anticipated resource management problems prevent you from using general-purpose algorithms and data structures. For example, linked data structures such as lists and trees are useful in a wide range of applications. Most C programmers learn to manage the memory for linked structures using the standard library functions **`malloc`** and **`free`**. A typically general implementation for **`malloc`** and **`free`** can be relatively slow — so slow that some real-time applications can't afford to use them. Some programmers use this slowness as an excuse to abandon linked structures altogether and use statically allocated global data structures instead. Unfortunately, programs with lots of global data are always brittle.

Just because an application can't afford to use general versions of **`malloc`** and **`free`**, that doesn't mean the application can't use linked structures at all. It only means it can't manage the memory for those structures using **`malloc`** and **`free`**. The program might

be able to use linked structures in conjunction with a less general memory management scheme that yields acceptable performance.

Before you get too hung up on avoiding the run-time overhead of certain language features, remember these basic principles:

- Choosing the right data structures and algorithms usually has a much greater impact on efficiency than localized optimizations do.

- In typical programs, 80% of the resources are used by only 20% of the code. (This is often referred to as the *80-20 Rule*. Some versions of the rule use slightly different ratios, such as 90-10.)

It's usually difficult to predict *a priori* which 10% or 20% of the code hogs the resources. As Bentley [1982] observes:

> Programmers are notoriously bad at guessing which parts of the code are the primary consumers of the resources. It is all too common for a programmer to modify a piece of code expecting a huge time savings and then to find that it makes no difference at all because the code was rarely executed.

Kernighan and Plauger [1976] stress the importance of measuring programs to figure out what's eating up the resources:

> The best procedure for obtaining efficient code is to choose a good algorithm, write a program that implements it as cleanly as possible, then measure it. The measurements will lead you directly to one or two routines that are worth making as efficient as possible. If they are clearly written and they hide their information properly, they will be easy to change. Sacrificing readability for efficiency earlier than this, while the bulk of code is being written, not only results in wasted effort but also leads to code that is hard to improve because it is hard to read.

In short, you should tune parts of your program only after you've made measurements that clearly identify what needs to be tuned. Bentley [1982], Meyers [1996], and Bulka and Mayhew [2000] echo these sentiments.

Of course, this doesn't mean you can ignore all concerns about resource management until you're ready to take measurements. If at all possible, you should base your design on preliminary performance estimates. However, you shouldn't use those estimates as a rationale for making inflexible policy decisions that permeate your code. Rather, you should try to confine each resource management decision to as little of the program as possible, preferably within a single class. Isolating such decisions makes it possible to tune the affected data structures and algorithms without disrupting the rest of the program.

Also, remember that the fastest parts of a program are those that are done before the program even starts. In general, you should:

- Catch errors during translation time (compiling and linking) rather than at run time. You should make the most of type checking and access checking, which means you should avoid using casts and **`void *`**.

- Simplify run-time computations by doing what you can at compile time.

- If you must do computations at run-time, shift them from the most time-critical parts to the least time-critical parts of the program (which is often, but not always, program start-up).

- When all else fails, try using C++ as just C.

With these general remarks in mind, let's look at some specific ways that run-time overhead can creep into C++ programs. As you will see, there are usually ways to reduce overhead without seriously compromising the design.

Parameter Passing

C++ normally passes arguments by value (just as C does). For example, given:

```
T v;
void f(T x);
```

the call **`f(v)`** copies argument **`v`** to parameter **`x`** just before transferring control to **`f`**.

If type **`T`** is a built-in type such as **`int`**, passing by value is pretty inexpensive — as inexpensive as it is in C. However, if **`T`** is a class type, passing by value copies **`v`** to **`x`** by invoking a copy constructor. Calling the copy constructor isn't necessarily expensive, but it can be very expensive. C++ offers you several ways to avoid that expense.

As in C, C++ lets you pass parameters by address. That is, you can declare **`f`** as:

```
void f(T *x);
```

For large objects, passing by address is clearly faster that passing by value. Unfortunately, you must then write the call as **`f(&v)`**, which changes its appearance. It also

changes the semantics of the call by making it possible for **f** to change the value of **v**. At the very least, you should use the **const** qualifier in the parameter declaration, as in:

```
void f(T const *x);
```

to ensure that **f** does not change the value of its actual argument.

For objects of modest size (about 8 to 16 bytes), it isn't always clear at the outset whether passing by address will actually be faster than passing by value. Switching between pass-by-address and pass-by-value poses a maintenance problem. Not only must you change the function declarations, as in:

```
void f(T x);  ⇨  void f(T const *x);
```

but you must also change all the function calls, as in:

```
f(v);  ⇨  f(&v);
```

The alternative is to declare **f** using a reference-to-**const** parameter, as in:

```
void f(T const &x);
```

This lets you write the call as **f(v)** (as if it were still passing by value), but yields the efficiency of passing by address. Again, the **const** qualifier ensures that **f** can't change the value of its actual argument, thus preserving the semantics of calling by value.

Passing by reference-to-**const** may have a surprising cost. When passing by reference-to-**const** the compiler may construct a temporary object holding a copy of the argument. When it constructs a temporary, passing by reference-to-**const** is slower than passing by value. Here's how it happens.

Normally, a constructor that can be called with one argument is a *converting constructor*. For example, the constructor **T(int)** in:

```
class T
    {
public:
    T();
    T(int);
    ...
    };
```

is a converting constructor. Given:

```
void f(T const &x);
```

then calling **f(10)** creates a temporary **T** object using the constructor **T(int)** and binds reference parameter **x** to that temporary. This is more expensive than passing a **T** by value.

An alternative is to declare **f**'s parameter as a reference-to-non-**const**:

```
void f(T &x);
```

This inhibits the compiler from creating temporary objects for parameter passing. (Actually, some compilers will create the temporary and bind the reference, but issue a warning.) Therefore, it lets you pass only **T** objects to **f**. Unfortunately, now the declaration permits **f** to alter the value of its actual argument.

If you are the author of class **T** as well as function **f**, then the best solution to this problem is to declare the constructor **T(int)** with the keyword **explicit**, as in:

```
class T
    {
public:
    T();
    explicit T(int); // a non-converting constructor
    ...
    };
```

A constructor declared **explicit** is a *non-converting constructor*. The compiler cannot use a non-converting constructor as an implicit conversion. For example, if **T(int)** is a non-converting constructor, then a call such as **f(10)** will no longer compile because the compiler cannot use **T(int)** to convert **10** to a **T** implicitly.

- Declare each constructor with the keyword **explicit** unless you need the constructor to be a converting constructor.

Using non-converting constructors doesn't make the code any more or less efficient that it would otherwise be. However, it makes the inefficiencies explicit, and avoids other nasty surprises.

Declaring the constructor does not completely prevent you from passing a temporary object as the parameter to **f**. It just prevents you from doing it inadvertently. You can still create a temporary explicitly, using any of explicit conversions:

```
f((T)10);               // C-style cast
f(T(10));               // function-style cast
f(static_cast<T>(10));  // new-style cast
```

This creates (and later destroys) a temporary, but it makes you go out of your way to do it.

Once you know that you really need to construct a **T** object, you can opt to construct a constant **T** object at namespace scope initialized with the value **10**, as in:

```
T const ten (10);
```

Then each call **f(ten)** merely binds a reference-to-**const** parameter directly to **ten**, without constructing or destroying a temporary object. The program will construct **ten** once at program startup, and destroy it only once at program termination. This is as cheap as it gets.

Temporary Objects

C++ compilers may generate temporary objects not only during parameter passing, but in various other situations as well, such as when returning values from function calls and during object initialization.

For example, suppose class **T** represents an arithmetic type with operators such as **+** and **-**. That is, you can write expressions such as:

```
T t, u, v;
...
t = u + v;
```

C++ translates the expression just above into a sequence of function calls:

```
T temp (operator+(u, v));   // initialize temp with sum
t.operator=(temp);          // copy temp to t
temp.~T();                  // destroy temp
```

Here, the compiler generates code to create a temporary object to hold the result of **operator+** before passing it on to **operator=**. The compiled program must also destroy the temporary when it's no longer needed.

Now, if **T** were a built-in type, such as **int** or **double**, the compiler would know enough to rewrite the code as:

```
t = u;
t += v;
```

thus eliminating the temporary object. Unfortunately, C++ makes no guarantees that this substitution works for class types, so if you want to eliminate the temporary, you must rewrite the expression yourself in terms of **+=**. That is, you should:

- Avoid generating temporaries during expression evaluation by writing:

  ```
  a = b op c;
  ```

  as:

  ```
  a = b;
  a op= c;
  ```

Here's another example involving temporary objects. The STL (Standard Template Library) that's now part of the Standard C++ library includes container classes, such as lists and queues, which work with iterators. An *iterator* is an object that "points" to an element in a container. You can use an iterator to visit each element in a container.

For instance, STL provides a template class **list<T>** for implementing lists with elements of type **T**. **list<T>** includes a public member **iterator** that's a class for declaring iterator objects that can traverse a **list<T>**. Here's some typical code that works with a **list<T>**, where **T** is **string**:

```
list<string> roster;
list<string>::iterator i;
...
for (i = roster.begin(); i != roster.end(); ++i)
    {
    // do something with *i
    }
```

A **list<string>** is an abstraction — you can't tell exactly how it's implemented. However, you can use an **list<string>::iterator** object, such as **i**, as a pointer to each element in the list. ***i** returns the list element that **i** "points" to, and **++i** "increments" **i** to the next list element.

Many programmers probably would have written the loop using **i++** instead of **++i**, just out of habit. If **i** were an **int** or a true pointer type, there would be no difference in the generated code for **i++** or **++i** when used as a stand-alone expression. However, when **i** has a class type with **operator++** as a non-inline function, **i++** actually yields poorer (bigger and slower) code because it generates a temporary object. Here's why.

The prefix expression **++i** generates a call of the form **i.operator++()**. A typical implementation of prefix **++** looks like:

```
iterator &iterator::operator++()
    {
    p = p->next;
    return *this;
    }
```

The call returns a reference to the incremented iterator without creating a temporary.

The postfix expression **i++** generates a call of the form **i.operator++(0)**. A typical implementation of postfix **++** looks like:

```
iterator iterator::operator++(int)
    {
    iterator i (*this);
    p = p->next;
    return i;
    }
```

The call returns a temporary copy of the iterator containing the value prior to incrementing.

Calling postfix **++** is not necessarily slower than calling prefix **++**. If the call ignores the old value of the iterator and postfix **operator++** is an inline function, the compiler may eliminate the temporary. If postfix **operator++** is an inline function, the compiler may avoid passing **0** as well. But, since prefix **++** never generates a temporary nor passes **0**, calling prefix **++** is never slower than calling postfix **++**. Therefore, when writing stand-alone increment expressions...

- Use **++i** in preference to **i++** no matter what type **i** is.

Ditto for the **--** operator.

Streamlining Initialization

Let's look again at the code for the loop:

```
list<string> roster;
list<string>::iterator i;
...
for (i = roster.begin(); i != roster.end(); ++i)
   {
   // do something with *i
   }
```

This definition:

```
list<string>::iterator i;
```

uses *default initialization.* That is, it calls a default constructor. This later statement:

```
i = roster.begin();
```

uses assignment to replace the default value. The program never uses **`i`** until after the assignment, so initializing **`i`** by default is wasted effort. You can avoid the overhead of default initialization by initializing **`i`** with the assigned value, as in:

```
list<string>::iterator i = roster.begin();
```

Unfortunately, even this initialization might produce a temporary:

- Initialization of the form **`T x = a`** is called *copy initialization*, and may create a temporary.

- Initialization of the form **`T x (a)`** is called *direct initialization*, and does not create a temporary.

Thus, you can assuredly avoid creating a temporary iterator by writing **`i`**'s declaration as:

```
list<string>::iterator i (roster.begin());
```

So here's the general advice:

- Avoid unnecessary default initialization. Initialize objects by direct initialization rather than by copy initialization whenever possible. Delay the declaration and initialization of an object until just prior to the first use of the object.

Here's a little more fine-tuning you can do. This loop calls **`roster.end()`** on each iteration:

```
list<string>::iterator i (roster.begin());
for (; i != roster.end(); ++i)
    {
    // do something with *i
    }
```

If **`list<string>::end`** is not an inline function, then the call contributes overhead to the loop. Of course, you can avoid the recalculation by rewriting the loop as follows:

```
list<string>::iterator i (roster.begin());
list<string>::iterator const end (roster.end());
for (; i != end; ++i)
    {
    // do something with *i
    }
```

Narrow Vs. Wide Interfaces

Always remember that simplicity leads to correctness, maintainability, and other good things. Therefore, you should build classes that are simple enough to do the job, and no more. However, a simple (narrow) class interface may incur performance penalties. You can avoid some of those penalties by adding more functions — widening the interface.

Consider a class for rational numbers (exact fractions). For example,

```
rational r1 (1, 3), r2 (4, 5);
```

defines **`r1`** with initial value **`1/3`**, and **`r2`** with initial value **`4/5`**. We can't make rational numbers look entirely as if they were built in, but we can come pretty close. This example implements a rational number as a pair of long integers representing the numerator and denominator:

```
// rat.h - rational number interface

class rational
    {
public:
    rational(long n = 0, long d = 1);
    rational &operator+=(rational const &ro);
    rational &operator-=(rational const &ro);
    ...
private:
    ...
    long num, denom;
    };
```

The constructor:

```
rational(long n = 0, long d = 1);
```

can be called with zero, one or two arguments:

```
rational r1;            // 0/1
rational r2 (9);        // 9/1
rational r3 (-1, 4);    // -1/4
```

Since it can be called with one argument, and it is not declared **`explicit`**, it is another example of a converting constructor.

For example, for **`rational r`**,

```
r = 3;
```

converts **`3`** to **`rational`** by calling the converting constructor, and passes that **`rational`** to the copy assignment. That is, the assignment generates:

```
r.operator=(rational(3, 1));
```

The converting constructor allows implicit conversions from any integral type to rational. The converting constructor can apply to the right-hand operand of all rational operators, such as **`+=`** and **`-=`**, as well as **`=`**. For example:

```
r = 3L;  // r = long int
r += 2U; // r += unsigned int
```

Unfortunately, each conversion creates a temporary. For example,

```
r = 3L;  // r = long int
```

translates more-or-less into:

```
rational temp (3L, 1L);
r.operator=(temp);
temp.~rational();   // destroy temp
```

The compiler may be able to generate code that avoids creating the temporary objects, but there's no guarantee that it can. You can help the program avoid creating and destroying temporary rational objects by implementing a wider interface for the rational number class. That is, you can define additional members and friends with parameter types chosen to reduce the need for implicit argument conversions. For example, in addition to the compiler-generated copy assignment:

```
rational &operator=(rational const &r);
```

you can declare:

```
rational &operator=(int i);
rational &operator=(long l);
```

and possibly others for other integral types, as needed. Then an assignment such as:

```
r = 3L;
```

translates into just:

```
r.operator=(3L);
```

Similarly, you can declare members:

```
rational &operator+=(int i);
rational &operator+=(long l);
```

to go with:

```
rational &operator+=(rational const &r);
```

In general, you should design classes with narrow interfaces. You should widen the interface only as needed to eliminate compiler-generated temporary objects at function calls. Wide interfaces generally increase code size and hinder maintenance. You shouldn't design wide interfaces without just cause.

Virtual Functions

C++ provides virtual functions as a way of implementing subtle differences in behavior among related types, while at the same time hiding those differences behind a common interface. Virtual functions employ dynamic (late) binding. That is, for a given virtual function call, the program can't decide which function it will call until it actually makes the call. Thus virtual functions, while powerful, also have a run-time cost. That cost is typically 2 to 4 instructions per call.

The cost of a virtual function call is usually small enough that you need not worry about it. However, a virtual call may appear in a time-critical part of the program where even this small overhead is unacceptable. For example, the call might appear inside a very busy loop:

```
for (i = 0; i < gazillion; ++i)
    {
    bp->f(i);       // f is virtual
    // other stuff
    }
```

In that case, you might look for ways to use a non-virtual call instead. (This example assumes that **bp** is declared with type **B *** , where class **B** is the root of an inheritance hierarchy and **f** is a virtual member function of **B**.)

If you have access to the source code for the class, you can try rewriting the function in question as a non-virtual function. However, there's a good chance this will adversely affect (break!) other parts of the program. Fortunately, C++ offers a simple notation for turning an individual call to a virtual function into a non-virtual call: if **f** is a virtual member function of class **T**, a call expression that refers to that function by its fully-qualified name **T::f** is an non-virtual function call.

For example, if you somehow know that **bp** actually points to a **B** object (rather than an object of a type derived from **B**), then you simply rewrite the call:

```
bp->f(i);       // virtual call
```

as:

```
bp->B::f(i);    // non-virtual call
```

The problem is, how can you know that **bp** actually points to a **B**? If there's a class **D** derived from **B**, then **bp** might point to a **D** object. Turning off the virtual call may make the call faster, but then it might call the wrong function.

One solution is to use *run-time type information* in a part of the program that's less time-critical than the loop. For example, if **bp** might point to either a **B** object or a **D** object (where **D** is derived from **B**), you can write:

```
if (D *dp = dynamic_cast<D *>(bp))
    {
    for (i = 0; i < gazillion; ++i)
        {
        dp->D::f(i);        // non-virtual call to D::f
        // other stuff
        }
    }
else // *bp really is a B
    {
    for (i = 0; i < gazillion; ++i)
        {
        bp->B::f(i);        // non-virtual call to B::f
        // other stuff
        }
    }
```

dynamic_cast is a built-in operator in C++. **dynamic_cast<D *>(bp)** attempts to convert the value of **bp** to type **D ***. If it succeeds (because **bp** points to an object with type **D** or a type derived from **D**), the cast returns a pointer to the complete **D** object surrounding ***bp**. Otherwise, it returns a null pointer.

The code above works fine if **D** is the only type derived from **B**. If ***bp** actually points to an **F** object, where **F** is derived from **D**, then the **dynamic_cast** expression:

```
D *dp = dynamic_cast<D *>(bp)
```

will still succeed (because an **F** is a **D**). Unfortunately, then the call:

```
dp->D::f(i);        // non-virtual call to D::f
```

will treat the **F** object as a **D** object. That is, it will call **D::f** instead of **F::f**. This might yield an erroneous result.

One way to solve this problem is to rewrite the code as follows:

```
if (typeid(*bp) == typeid(D))
    {
    D *dp = dynamic_cast<D *>(bp);
    for (i = 0; i < gazillion; ++i)
        {
        dp->D::f(); // non-virtual call to D::f
        // other stuff
        }
    }
else if (typeid(*bp) == typeid(B))
    {
    for (i = 0; i < gazillion; ++i)
        {
        bp->B::f();// non-virtual call to B::f
        // other stuff
        }
    }
else // *bp is neither a B nor a D
    {
    for (i = 0; i < gazillion; ++i)
        {
        bp->f();    // virtual call
        // other stuff
        }
    }
```

`typeid` is another built-in operator in C++. **`typeid(e)`** returns a **`typeinfo`** object that describes the dynamic type of expression **`e`**. **`typeinfo`** is a class defined in the standard header **`<typeinfo>`**. An expression such as:

```
typeid(*bp) == typeid(D)
```

is true if **`bp`** points to an object whose type is exactly **`D`**, not something derived from **`D`**.

The code holds up even if the type hierarchy is extended with new types. The if-clause handles objects of type **`B`** using non-virtual calls to **`B::f`**.. The else-if clause handles objects of type **`D`** using non-virtual calls to **`D::f`**. The trailing else-clause handles all other types derived from **`B`** using a virtual call. That virtual call may be slow, but it will produce correct results.

This coding technique trades space for speed. It moves the dynamic binding decision to a point outside the loop at the expense of duplicating the code for the loop. This is not something you want to do very often, but it probably beats the alternatives in this case.

It's a localized hack that introduces local complexity, but it preserves the overall design of the program.

Operators New and Delete

Some embedded systems developers avoid dynamic memory because dynamic allocation and deallocation are allegedly slow. However, dynamic allocation often uses memory more efficiently than static allocation. A statically allocated object always occupies memory, whereas a program can release the memory used by a dynamically allocated object and reuse that memory for another object.

Most parts of a real-time system are not time-critical. You need be concerned about the speed of dynamic allocation and deallocation only in the time-critical parts of your software. Moreover, C++ offers techniques to speed dynamic allocation so that it can meet the demands of all but maybe the most time-critical applications. The techniques involve redefining operators **`new`** and **`delete`**.

A new-expression actually allocates memory for non-array objects by calling an allocation function named **`operator new`**. Every C++ environment provides a default implementation for a global function **`operator new`**, declared in the C++ standard header **`<new>`** as:

```
void *operator new(std::size_t n) throw (std::bad_alloc);
```

The argument to **`operator new`** is a value of type **`std::size_t`** representing the size (in bytes) of the requested storage. The throw-specification:

```
throw (std::bad_alloc)
```

at the end of the function heading stipulates that, if the allocation fails, the function shall throw an exception only of the standard class **`std::bad_alloc`** derived from the standard class **`std::exception`**.

`std` is the namespace that contains nearly all the standard library components. Even though it is not declared in namespace **`std`**, the global **`operator new`** is declared in terms of the names **`size_t`** and **`bad_alloc`** which are declared in **`std`**. For the remainder of this discussion, I shall refer to **`size_t`** and **`bad_alloc`** without the **`std::`** prefix. That is, you should assume that the using-directive:

```
using namespace std;
```

is in force from here on.

An expression such as:

```
p = new T;
```

translates into something (sort of) like:

```
p = (T *)operator new(sizeof(T)); // allocate
p->T();                           // initialize
```

That is, it acquires memory for the object by calling an **`operator new`**, and then applies a constructor to that storage. (The latter expression — an explicit constructor call — is not something you can actually write in C++.)

A delete-expression deallocates memory for non-array objects by calling a deallocation function named **`operator delete`**. Each C++ environment provides a default implementation for a global function **`operator delete`**, declared in **`<new>`** as:

```
void operator delete(void *p) throw ();
```

The empty throw-specification:

```
throw ()
```

indicates that the function shall not throw any exceptions.

An expression such as:

```
delete p;                         // for T *p
```

translates into something resembling:

```
if (p != NULL)
    p->~T();                      // destroy
operator delete(p);               // deallocate
```

That is, if **`p`** is not null, the delete-expression applies a destructor to **`*p`**, and the releases the storage for **`*p`** by calling an **`operator delete`**. The explicit destructor call is something you can write in C++.

In many C++ development systems, the default implementation for **`operator new`** calls **`malloc`**, and the default implementation for **`operator delete`** calls **`free`**.

However, if the default allocation/deallocation algorithm isn't right for your application, you can write your own versions of operators **new** and **delete**. Somewhere in your program, you can define functions such as:

```
void *operator new(size_t n) throw (bad_alloc)
    {
    void *p = my_allocator(n);
    if (p == NULL)
        throw bad_alloc();
    return p;
    }

void operator delete(void *p) throw ()
    {
    my_deallocator(p);
    }
```

where **my_allocator** and **my_deallocator** represent your customized memory management algorithm(s).

Writing your own global memory manager can be very challenging because a global memory manager must handle memory requests of widely varying sizes. There are companies in the business of writing replacement dynamic memory managers, and you should consider buying one before you write your own.

In many applications, the majority of dynamically allocated objects tend to be of just a few types. You may be able to achieve significant performance improvements by using special-purpose allocation and deallocation functions for just those few heavily-used types, while still using the library's general-purpose allocation and deallocation functions for all the other (less often used) types. C++ lets you implement special-purpose memory managers for objects of a given class by defining **operator new** and **operator delete** as members of that class. For example:

```
class T
    {
public:
    void *operator new(size_t n) throw (bad_alloc);
    void operator delete(void *p) throw ();
    ...
    };
```

defines class **T** with class-specific versions of **operator new** and **operator delete**. Thereafter, a new-expression such as:

```
pt = new T;
```

allocates memory using **T::operator new** rather than the global **operator new**. Likewise:

```
delete pt;
```

deallocates memory using **T::operator delete**. Both **T::operator new** and **T::operator delete** are static member functions, whether or not declared with the keyword **static**. They cannot be virtual.

Allocation and deallocation for scalar types (built-in arithmetic types, enumerations and pointers), as well as class types without member operators **new** and **delete**, always use the global **new** and **delete**. If class **T** has its own **new** and **delete**, you can still request the global **new** and **delete** for **T** objects by explicitly using **::**, as in:

```
p = ::new T;        // calls ::operator new
...
::delete p;         // calls ::operator delete
```

Available Lists

One of the simplest, most effective strategies for speeding up dynamic allocation for a class **T** is to maintain memory for free **T** objects in a linked list. This is particularly easy to implement if class **T** already has a member of type **T *** . For example:

```
class T
    {
public:
    void *operator new(size_t) throw (bad_alloc);
    void operator delete(void *) throw ();
    static void acquire(size_t);
    static void release();
    T();
    T(int);
    ~T();
    ...
```

```
private:
    T *next;
    // ... other non-static data ...
    static T *available;
    };

T *T::available = NULL;
```

available points to the head of a list of **T** objects that are not currently in use. **T**'s **operator new** allocates storage for a **T** object simply by taking the first object from the head of the available list:

```
void *T::operator new(size_t n) throw (bad_alloc)
    {
    T *p = available;
    if (p == NULL || n != sizeof(T))
        throw bad_alloc();
    available = available->next;
    return p;
    }
```

T's **operator delete** discards objects by inserting them at the head of the list:

```
void T::operator delete(void *p) throw ()
    {
    if (p != NULL)
        {
        T *xp = static_cast<T *>(p);
        xp->next = available;
        available = xp;
        }
    }
```

How does the program initially populate the available list? One simple and efficient approach is to allocate an array of **T** objects at program start-up, and thread the array elements into a list:

```
void T::acquire(size_t n)
    {
    available = ::operator new(n * sizeof(T));
    T *p = available;
    for (; p < available + n - 1; ++p)
        p->next = p + 1;
    p->next = NULL;
    }
```

New and Delete for Arrays

In early dialects of C++, allocating an array of **T** objects always used **::operator new** even for a class **T** that defines its own **operator new**. That is:

```
p = new T [n];
```

ignored **T::operator new** and used **::operator new**. Similarly, deleting that array using:

```
delete [] p;
```

ignored **T::operator delete** and used **::operator delete**.

Standard C++ now provides a separate set of dynamic memory management functions for arrays of objects:

```
void *operator new[](size_t n) throw (bad_alloc);
void operator delete[](void *p) throw ();
```

Thus, an expression such as:

```
p = new T [n];
```

uses an **operator new[]** instead of an **operator new**, and

```
delete [] p;
```

uses an **operator delete[]** instead of an **operator delete**. We often refer to **operator new[]** and **operator delete[]** as *array new* and *array delete*, respectively.

You can define **operator new[]** and **operator delete[]** for individual classes, either with or without **new** and **delete** as well. For example, if you define:

```
class T
    {
public:
    void *operator new(size_t) throw (bad_alloc);
    void *operator new[](size_t) throw (bad_alloc);
    void operator delete(void *p) throw ();
    void operator delete[](void *p) throw ();
    ...
    };
```

then

```
p = new T [n];
```

uses **T::operator new[]** instead of **::operator new[]**.

New with Placement

You can overload **operator new** and **operator new[]**. In particular, you can declare functions with these names that have additional parameters after the first parameter of type **size_t**. For example, in addition to the usual:

```
void *operator new(size_t) throw (bad_alloc);
```

the Standard C++ library provides:

```
void *operator new(size_t n, void *p);
```

to allocate memory from the pool addressed by **p**. You can call this alternative **operator new** using the *placement syntax*, which supplies additional arguments immediately after the keyword **new**, as in:

```
p = new (pool) T;
```

The call passes **sizeof(T)** as the first argument (of type **size_t**), and **pool** as the second argument (of type **void ***).

The general form of a (non-array) new-expression is:

new (p_2, ... p_i) T (a_1, ... a_j)

where:

- **(p_2, ... p_i)** are the 2nd through **i**th arguments to **operator new** (the first argument is always the size of the requested storage)
- **T** is the type of requested object
- **(a_1, ... a_j)** are the arguments to a **T** constructor

You can use placement arguments with **operator new[]**. You can also use placement arguments with **operator new** and **operator new[]** as class members.

The general form of an array new-expression is:

new (p_2, ... p_i) T [n]

where:

- **(p_2, ... p_i)** are the 2nd through **i**th arguments to **operator new[]** (again, the first argument is always the size of the requested storage)
- **T** is the element type of the requested array
- **n** is the number of elements in the requested array

Array allocation always uses the default constructor.

You can use **new** with placement to reduce the cost of storage allocation and deallocation. For example, suppose some part of your program deletes a pointer **p** pointing to a **T** object, and then turns around almost immediately and allocates another **T** object, as in:

```
delete p;
...
p = new T;
```

If your performance measurements indicate that the overhead of deallocating and reallocating these objects is too costly at this point in the program, you might be able to reduce that overhead by rewriting the code as:

```
p->~T();    // destroy an object
...
new (p) T;  // construct a new one
```

This code destroys the object addressed by **p**, but does not call a deallocation function to discard the storage for that object. It later uses **new** with placement to construct a new **T** object in the storage formerly occupied by the **T** object that was. As long as the distance

between these two statements is short, and there's a comment to explain this little hack, this is probably acceptable style.

If you would like to prevent your application from allocating objects of a certain type **T** on the free store, simply declare the operators **new** and **delete** for **T** as private:

```
class T
    {
public:
    ...
private:
    void *operator new(size_t);
    void operator delete(void *);
    ...
    };
```

You may wish to do the same for array **new** and **delete**.

Closing Remarks

Embedded systems have widely varying needs for resource management. Consequently, you must evaluate the needs of each system individually. In general, good programs focus on abstract data types and algorithms. They confine resource management decisions to relatively small parts of the program (such as individual classes).

C++ lets you redefine many facilities to improve performance significantly. You should keep an open mind about most C++ facilities – don't reject a particular C++ facility just because the default implementation is too inefficient for some part of your application. Look for ways to fine-tune that facility before dismissing it.

References

- Bentley [1982]. Jon L. Bentley, *Writing Efficient Programs*. Prentice-Hall.
- Bulka and Mayhew [2000]. Dov Bulka and David Mayhew, *Efficient C++*. Addison-Wesley.
- Kernighan and Plauger [1976]. Brian W. Kernighan and P. J. Plauger, *Software Tools*. Addison-Wesley.
- Meyers [1996]. Scott Meyers, *More Effective C++*. Addison-Wesley.