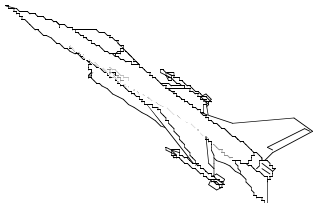
Can software development be elevated from the *Microsoft* * style?

Oleg Kiselyov
CIS, Inc
3401, E.University Dr., Suite 104 Denton TX 76208
oleg@ponder.csci.unt.edu, oleg@unt.edu, http://mozart.compsci.com/~oleg/ftp/

*Microsoft epitomizes bloated and unsafe programming combined with poor design and trading of quality for speed and market leverage. This spoils even good ideas they occasionally have. This paper is an eclectic study of (mostly C++) programming techniques, with **lots** of "good" and "bad" code snippets, and pessimistic conclusions. The bad snippets are taken from OpenDoc, MacTech magazine, and code found in trade rags and on the net. To contrast good and bad, the snippets are rewritten in a better style. Moreover, the paper shows off and expounds on a few immensely powerful and safe programming techniques, like nested functions, lazy objects, stealing of the body, iterators in a local context. Deep obscurity of these techniques tells however that resistance is futile and we all will be assimilated.*



"An ugly plane won't fly" - an aviator's saying

* The use of Microsoft here is for hyperbole only and in no way should be construed as an insult.

Pointers vs. References

This is a fairly well beaten track. Before I swerve off it though, let me add a few more lashes. Besides, there are still too many dangerous and unnecessary pointers in C++ code out there. Indeed, in *very* many situations where pointers are commonly used:

- passing big objects to a function: `void foo(BigStruct* p);`
- emulating "output" function parameters (by-reference):
```
void swap(int* a, int* b);
```
- referring to out-of-body object items
```
class DrawEditor {
  COrderedList* fEmbeddedFrames; };
```

pointers can be replaced by references. And they should be replaced by references wherever possible, for a good reason that references are harder to mess up. Indeed, you cannot forget to initialize a reference (it simply can't be created uninitialized), you cannot inadvertently "drop a star" like in

```
void foo(int *p)
{... *p +=1; p += 1; }
```

(where both operators are syntactically correct and semantically valid; chances are only one is meant (often the first)). Pointers have an edge in that they can possess a special value 0 (NULL, nil), which often means some special circumstance (say, the function generated no output, or an input parameter is omitted). Still, this can be emulated with references: `void foo(int& a) {a = 1;} foo(int());` calls `foo()` squarely for its side-effect, disregarding the output variable.

Pointers to a function make up a slightly special case. In C++, this case has to be even more special:

```
double minimizer(double x0,
   double (*fn_being_minimized)
          (const double x))
{
  double f0 =
     fn_being_minimized(x0);
  ...
}
```

Fig. 1a. A typical usage of a function pointer

```
class Minimizee
{
public:
  virtual double operator ()
     (const double x) = 0;
};
double minimizer(double x0,
   Minimizee& fn_being_minimized)
{
  // exactly the same usage
  double f0 =
     fn_being_minimized(x0);
  ...
}
```

Fig. 1b. A better alternative: function class

'Function' classes provide a safer, and better alternative. The abstract 'function' class defined on Fig. 1b. can be fleshed out as follows:

```
void foo(void)
{
  class the_minimizee :
    public Minimizee
  {
    double a;
  public:
    double operator ()
      (const double x)
       {return sqr(x*x - a);}
   the_minimizee
    (const double _a) : a(_a) {}
  };
  cout << "min value " <<
   minimizer(the_minimizee(10))
   << endl;
}
```

From the operational point of view, `function` class is almost identical to function pointer. In a sense, a function class is a syntactic wrapper around a function pointer, which is tucked away into a `vtable`. Since the pointer itself is hidden, however, there is no way one can mess it up, leave uninitialized, or use inappropriately. Moreover, a `function` class gives an opportunity to pass some "environment" for the callback function, which is much more elegant a solution than ubiquitous `RefCon`.

Are pointers really necessary? When C is better than Java

Daring and heretical as it sounds, *sometimes* pointers *are* indispensable. For one thing, to emulate a late (dynamic) binding. Let's take `iostreams` as an example. One can create a dummy (empty) `istream` object, associate it with a file buffer, re-attach it later to a string buffer (and later make it share a buffer with another stream). Thus, stream buffer actually acts as a "virtual member" of the `istream` class. Using pointers (to an abstract base class, `streambuf` in our example) is the only way in C++ to emulate virtual (or virtual+dynamic) data items.

The most common situation where pointers are really necessary, and where they shine is a *serial* access to collections, mostly from within *loops*. Serial (as opposed to random) access means getting hold of elements of a collection in some sequential order, one *after* another, rather than hopping around. It is surprising to realize that many algorithms do not actually require random access. Sequential access is obviously faster than a random one: this is why a C program can perform better than an equivalent Fortran (Java) code. As an example, let's consider a (overhyped as usual) claim "Java will also make it easy for developers to distribute new technologies, such as video compression algorithms, without having to target a specific computer architecture" (http://www.sun.com/sunworldonline/swol-03-1996/swol-03-multimedia.html), and contrast

it with reality. As far as video/image processing is concerned, one of the most common operations is color conversion, say, from RGB to YUV. In Fortran/Java, one can do that in the following generic way:

```
Pixel pix_matrix[] =
   new Pixel[N*M];
for(i=0; i<N*M; i++)
  pix_matrix[i] =
   rgbtoyuv_pixel(pix_matrix[i]);
```

Let's take a careful look at how it works: each evaluation of `pix_matrix[i]` involves:

```
assert( 0 <= i < N*M );
return &pix_matrix[0] +
        i*sizeof(Pixel);
```

Depending on the `sizeof(Pixel)`, the multiplication may be non-trivial. This overhead – two comparisons (beside a comparison in the loop termination condition) plus one multiplication – is repeated for *every* image pixel!

However, a pixel-by-pixel operation like the color-space conversion requires merely a serial access to the array of pixels: once you are done with one pixel, you go on to the *next* one. Thus a random access to pixels is totally unnecessary. With pointers, the conversion above would look like

```
const Pixel * const pp_end =
    &pix_matrix[0] + M*N;
for(register Pixel * p =
&pix_matrix[0]; p < pp_end; p++)
  rgbtoyuv_pixel(*p);
            // passing Pixel&
```

The overhead here is merely a single addition and a comparison. As to the claim of using Java for writing video codecs (which should be able to compress/decompress at least 24 frames per second), it is a clinical case of caffeine-induced hysteria.

Many linear algebra operations on vector and matrices do not actually require random access either. Even a matrix inverse can be done with a serial access only (e.g., LinAlg.shar package, /info-mac/dev/lib/lin-alg-cpp.hqx, ftp://replicant.csci.unt.edu/pub/oleg/LinAlg.cpt.hqx)

Efficient as pointers are, they are not safe. A pointer does not care if it points somewhere within array's boundaries or beyond them, or if it points to any sane place at all. STL has attempted to tame pointers by "casting" them into iterators. Of course, an iterator is as safe as it is implemented within a particular class. Still, there is an opportunity now to make a *safe pointer*, which looks and feels just like the regular one, but always points to a valid location. With iterators, one can also limit a set of commands a pointer obeys (for example, forbid steps backward). Another approach to safe pointers is *streams*:

```
void write_pixmatrix(const IMAGE&
image, EndianOut& file)
{
   // Write the entire pixmatrix in one chunk
Image_istream im_stream(image);
while( !im_stream.eof() )
  file.write_byte(
    im_stream.get());
}
```

Note that the entire iteration in the example above is inlined; it also requires less sanity checking overhead than a typical STL iterator. It has to be stressed that `im_streams` (like regular streams) are safe: even if you "forgot" to check for `eof()`, the program will not enter into an infinite loop or garble memory. One can well enlist regular `iostreams` for the same purpose (with a slightly higher overhead). Again, in the code above, there is only one check per iteration and all operations on the `im_istream` are done inline. Thus the snippet runs just as fast as a "traditional" code, only safely.

Finally, here is one more example where the wild beast of `char *` would have helped:

```
void
SCScriptsMenuHandler::MakeBallonData
(Str255 inHelp, char * ioBuffer)
{
  Int16 mark, data;
  Int32 zeros = 0;
  mark=2;
  if(*inHelp == 0)
  {   // no data, skip the item
    data = 0x0100;
    ::BlockMoveData(&data,
&ioBuffer[mark], sizeof(Int16));
    mark += sizeof(Int16);
  }
  else {
    data = 0x0001; // direct string type
    ::BlockMoveData(&data,
&ioBuffer[mark], sizeof(Int16));
    mark += sizeof(Int16);
            // write out the string
    ::BlockMoveData(inHelp,
       &ioBuffer[mark],1+*inHelp);
    mark += 1+*inHelp;
      // write out three zeros for
      // the other strings
    ::BlockMoveData(&zeros,
       &ioBuffer[mark], 3);
    mark += 3;
  }
  // align buffer to an even word boundary
  if( mark & 0x0001 ) ++mark;
 ::BlockMoveData(&mark,ioBuffer,
    sizeof(mark));
}
```

Fig. 2a. Original code from "Attaching a Scripts Menu," MacTech magazine, v.12, No.2, Feb. 1996, p. 65

```
void
SCScriptsMenuHandler::MakeBallonData
(Str255 inHelp, char * ioBuffer)
{
      // skip the first word for now
  char * cp = ioBuffer +
              sizeof(Int16);
  if(*inHelp == 0) // no data, skip
    *((Int16*)cp)++ = 0x0100;
  else {
            // direct string type
    *((Int16*)cp)++ = 0x0001;
            // write out the string
      //memcpy(cp,inHelp,1+inHelp[0]);
      //would've been better: inlined in CW
   ::BlockMoveData(inHelp,cp,
       1+*inHelp);
    cp += 1+inHelp[0];
      // write out three zeros for the other
      //strings
    *cp++ =0; *cp++ =0; *cp++ =0;
  }
  // align buffer to an even word boundary
  if( (int)cp & 1 )
   *cp++ = 0;
  *((Int16*)ioBuffer) =
    cp - ioBuffer;
}
```

Fig. 2b. Optimized code using pointers: 72 bytes smaller with all operands in registers

In the original code, Fig. 2a, a system trap (`BlockMoveData`) is used to put mere **two** bytes into memory! One cannot leave it like that. Note, that ioBuffer seems to be word-aligned. Indeed, if one takes a special step to align the end of buffer, its beginning should be aligned, too. So, one can use `Int16 *` ptr to write the flags. The optimized code, Fig. 2b, does `BlockMoveData` only once, to move a string body. Moreover, if one uses `memcpy()`, which is inlined by CodeWarrior, then the entire function on Fig. 2b is a leaf: calls no functions, makes no traps.

In-line vs. out-of-line construction of objects

When an object is constructed and destroyed within the same function, it can be allocated on stack: '**new**' is *really* unnecessary then. Here is a somewhat extreme case of a "transient" object:

```
case item_Go:
   if( is_flying )
     is_flying = FALSE,
     ModelessDialog::
      ControlItem(*this,item_Go).
      set_title("\pGo");
   else
     is_flying = TRUE,
     ModelessDialog::
      ControlItem(*this,item_Go).
```

```
        set_title("\pStop");
    image_3d_view.force_redraw();
```

`ControlItem` object is created and destroyed within the same statement, we don't even care to give it a name. The object is needed solely to put a new "face" on a dialog, and make a small request in this face. Thus `ControlItem` is a *reference* or *interface object*, providing a new view, facet to the existing data. Note, `ControlItem` is a simple structure, which does not have any dynamic data to allocate/free.

When an object is constructed and destroyed within the same object, it can be *incorporated*. And it is better be: C++ FAQ (http://osiris.sund.ac.uk/online/C++/cplus.html) has a special paragraph on this:

"Q101: Should class subobjects be ptrs to freestore allocated objs, or contained?

"A: Usually your subobjects should actually be *contained* in the aggregate class (but not always; *wrapper* objects are a good example of where you want a ptr/ref; also the N-to-1-uses-a relationship needs something like a ptr/ref)."

The FAQ then goes on to explain, why fully contained subobjects have better performance than ptrs to freestore allocated subobjects:

"- extra layer to indirection every time you need to access subobject

"- extra freestore allocations (`new` in ctor, `delete` in dtor)

"- extra dynamic binding

"Thus fully-contained subobjects allow significant optimizations that wouldn't be possible under the subobjects-by-ptr approach (this is the main reason that languages which enforce reference-semantics have *inherent* performance problems)."

It seems that OpenDoc designers should have listened to that advice. Excessive wrapping of pointers is one of the reasons OpenDoc is rather sluggish, and a big memory hog. For example (a `DrawEditor` sample part)

```
class DrawEditor {
  COrderedList*
   fEmbeddedFrames;
};
```

where `COrderedList` is declared in `samplecollections.h` as

```
class COrderedList {
public:
  COrderedList();
  COrderedList(COrderedList
*list);
  ODBoolean    IsEmpty() const;
  void         AddBefore(const
void* existing, void* value);
  void*        First() const;
                        // snipped
private:
    // which is probably just head ptr,
    // tail ptr, and the number of elems
  LinkedList          fList;
};
```

`DrawEditor` code then has to zero out the `fEmbeddedFrames` pointer in the constructor, dispose of the object if necessary in the destructor, and assign the pointer a `new COrderedList` object in an `init()` method, thus allocating only a few bytes (`sizeof(LinkedList)`) on heap! All of this would not have been necessary had the list been just a part of the `DrawEditor` object. It goes without saying how unsafe `COrderedList` is (forcing one to cast list elements' contents to `void*`)....

Sub-object incorporation can be done even in rather unfavorable circumstances, for example, when a late binding is unavoidable (see Appendix 1).

If an object is transient in nature, it is usually better to allocate it locally, on stack, and let it die when the current scope is over. One can also *define* an object class locally, within the current function's scope, which gives rise to ***nested functions*** and ***clauses***. They are indeed possible in C++, complete with name scoping, access control, etc. The local classes are very useful as *iteratee*s to be passed to an iterator, see below for more detail. Since local classes are declared (and belong to) the scope of their "outer" function/method, even the class' name (type) is not visible outside of that function, which prevents name conflicts. Note that nested methods (which are actually nested functions) are compiled inline, unless they are virtual. For a real (though somewhat absurd) example of ***double-nested*** classes, see http://mozart.compsci.com/~oleg/ftp/c++-digest/more_nested_func.txt

Copy semantics – a bane of C++

Too often too many intermediary (temporary) objects are constructed, only so that they can be assigned to other objects. Many copy-assignments can *easily* be avoided, without resorting to a full-blown reference counting and/or garbage collection.

The problem is manifest when a function has to build and *return* a complex object. Constructing of an object is a ctor's job, and better left to him. However, consider a situation when one comes across a brand new way to build an object, but the interface to a class library is already frozen, so there is no way to add a new constructor to the class. This situation is quite common: for example, loading an image from some (new) file or building a test matrix of some new particular kind. In any case, it is a *bad* idea to return thus constructed complex object as the function result, as in:

```
Matrix foo(const int n)
{ Matrix foom(n,n);
  fill_in(foom); return foom; }
Matrix m = foo(5);
```

When `foo(5)` is called, it creates a `Matrix foo::foom`, fills it in, copies it onto stack as the return value, and destroys `foo::foom`. The return value from `foo(5)`, a matrix, is then copied over to `m` (via a copy constructor). After that, the return value on stack is destroyed. Thus matrix constructors are called 3 times, and the destructor 2 times. For big matrices, it may be very expensive to construct/copy/destroy objects, especially over again. *Some* optimized compilers can cut down on one copying/destroying; still it leaves at least two calls to a constructor. LazyMatrices (aka promises) (see below) can construct `Matrix m` *in place*, with only a *single* call to a constructor.

Thus the first principle of returning complex objects from functions is: don't do it, let a constructor construct. This of course assumes that one has control over the interfaces/library, and can see far enough. To help differentiate among various constructors that take the same number of arguments, it is a good idea to use an "action code", for example,

```
IMAGE blown_out(IMAGE::Expand,
                Test_image);
```

or

```
Matrix haar = haar_matrix(5);
Matrix unit(Matrix::Unit,haar);
Matrix haar_t(Matrix::Transposed,
              haar);
Matrix hth(haar,
     Matrix::TransposeMult,haar);
Matrix hht(haar,
     Matrix::Mult,haar_t);
```

The true solution to a problem of returning an object however is a **lazy object** (a promise). That is, rather than return an object, you pass out a mere *recipe* how to make it. The full object would be rolled out only when and where it is needed. For example,

```
IMAGE map = FractalMap(order);
```

`FractalMap` is a class, not a simple function. No temporary image is ever constructed: the object `map` is filled out right in place, without moving a single pixel. A `FractalMap` constructor is actually quite trivial: all it does is filling in slots of a small `LazyImage` object, which specifies the image dimensions. The real job is done by a recipe, a virtual `FractalMap`'s function. The recipe is clearly easy to amend if one so wishes, by subclassing from `FractalMap` and overriding that virtual recipe, or other virtual functions (say, a random number generator) it is using. Since the recipe is called *after* the object is constructed (and passed, actually), overriding of virtual functions *works*.

The problem of returning of an object has seemingly one more solution: if it is so expensive to return an object itself, why not to return a pointer to it? As widely spread as it is, this is hardly cool. Obviously the object itself must be allocated on heap. That means the object has to be *explicitly* deleted later. Figuring out when it is safe to delete an object, and making sure the pointer is not used after the object is disposed of is often a tough job. Note that unlike "local" objects created through recipes, etc., heap objects are not automatically deleted if an exception is thrown.

Still, there are situations when one is compelled to return a pointer to a globally allocated object. For example, the object (the pointer, actually) is supposed to be put into a list. One can play it safe even then: for example,

by making it impossible to use the pointer for anything else but putting it into a specific list. Also, if the object must be allocated on heap, make it illegal to create it on stack. Thus any object pointer one can possibly get hold of is guaranteed to be a heap pointer, and is good only for adding to the list (and/or performing a very limited set of operations, cast being not among them). Surprisingly, all this can be accomplished with *no* overhead, as a code in Appendix 2 shows. It is a snippet from a real TIFF image writer.

Body and faces: polymorphism with a *compile* time type checking

A weak form of polymorphism – an objects with several faces to reply to the same message differently – is easily implemented through subclassing and virtual functions. But how about a collection of data changing its interfaces during its lifetime so that not only they reply to the same message differently, but can even take different messages? In short, can C++ support both a larva and a butterfly it turns to? And the worm will not be forced to fly before its time? As it turns out, it is possible, with a technique one could call "stealing of the body" (aka *passing the buck*). Here is a characteristic example: sound processing. Sound can be represented in a variety of ways, for example, as sampled voltage from a microphone (so-called PCM, pulse-code modulation), differences between two adjacent sample voltages, (DCM or ADCM, delta-code modulation), and the FFT of the samples. All these representations have the same number of data items, but their meaning is different. They also need different procedures, to, say, play a sound back. Certainly there are several ways to model "same data, different meaning" in C++ and to easily convert among them. The most obvious technique is

```
class Sound
{
  enum { PCM_type, ADCM_type,
         FFT_Type} type;
  const int no_samples;
  float * samples;
public:
  Sound() { record it somehow;
            type=PCM_type; }
  void do_adcm(void) {
    if( type == PCM_type )
      do_pcm_to_adcm();
    else if( type == FFT_Type )
      do_fft_to_adcm();
    type = ADCM_type; }
  void do_pcm(void) {
    if( type == .... )
       blah blah blah; }
};
```

This is a straightforward emulation of *really* polymorphic, dynamic objects, as found in such languages as Prolog, Lisp, PostScript, etc. These objects must have a special tag telling which incarnation they are presently in; all their methods have to check the tag and act accordingly. While this run-time type checking is fully intended in these languages, it is not very cool as far as C++ is concerned. Besides, there *is* a better way:

```
class Samples   // This is a "data" class: declares
{               // data without any "functionality"
  const int no_samples;
  float * const samples;
public:
  Samples(const int _no_samples)
    : no_samples(_no_samples), samples(new float[_no_samples]) {}
};

class Sound
{
  Samples * sample_data;
protected:
  Samples& q_samples(void) const { assert( sample_data != 0);
                                   return *sample_data; }
public:
          // Here's where the real fun begins:  A copy constructor rips off another
          // object and makes it a zombie!
  Sound(Sound& another_sound)
    { sample_data = another_sound.sample_data;
      another_sound.sample_data = 0; }
};
          // These are "functionality" classes which  dress the data in different ways
class ADCMSound;
class PCMSound : public Sound
{         // the class has no private data!
public:
  PCMSound(Recorder& recorder) : Sound(...)  { record(); }
  PCMSound(ADCMSound& adcm_samples) : Sound(adcm_samples)
    { do_adcm_to_pcm(q_samples()); }
  void play_back(void);
  void write(const char * file_name);
};

class ADCMSound : public Sound
{
public:
  ADCMSound(PCMSound& pcm_samples) : Sound(pcm_samples)
    { do_pcm_to_adcm(q_samples()); }

  void write(const char * file_name);
};
```

Converting among different representations is trivial:

```
PCMSound
pcm_sound(some_recorder);
ADCMSound adcm_sound(pcm_sound);
```

etc. You can `write()` into a file from either representation (of course, ADCM is more efficient). But if you want to play the sound back, you have to convert `ADCMSound` to `PCMSound` first (because speakers usually prefer dealing with PCM).

The "sound type" checking is mostly done at compile time. That is, the compiler itself can figure out which `write()` method to call. Error checking is also static: In the first, "dynamical", approach, it is the `Sound::play_back()` method itself who has to check object's tag and holler unless `type == PCM_type`. In the second approach, only `PCMSound` has a `play_back()` method. Therefore it is simply impossible to `play_back()` on a `ADCMSound` object. Since C++ is not a dynamic language

(well, it was not designed to be) there is some price to pay: after

```
ADCMSound adcm_sound(pcm_sound);
```

the `pcm_sound` object becomes a zombie, while the object's name sticks around (but must not be used!). In a true dynamic language (e.g., PostScript), one can dispose of both an object and its name.

Natural and unnatural iterators

The word 'iterator' has become rather strongly associated with STL. Although STL's iterators do not actually iterate: they are iterated upon. A better name would be an accessor, or iteratee. The most common, alas, use of STL's and similar iterators is in an *unnatural* iteration like the following:

```
for(x.first(); x.good();
    x.next())
  { process(x.current()) }
```

Note that `x` (an "iterator") has to maintain some status information: which element of a collection it points to, and when it is "good". Each iteration thus involves at least three checks to see if the iterator `x` is still good:

- before entering the body of the loop
- in accessing the current element
- before attempting to advance the iterator

With a help of pointers, one can cut down on a few checks:

```
for(Elem * i = x.first(); i != 0;
    i = x.next())
  { process(*i); }
```

Sample code in OpenDoc release 1.0 provides numerous examples of iterations like that. As efficient as it looks, the solution with pointers opens a pandora box: one can now change the pointer `i` inadvertently in the body of the loop, and nothing could prevent it. Also, there still remain a few extra checks: `x.next()` should always test if `x` is within the collection (and return `0` if not); the loop then has to check again if `i` is not a zero pointer. This overhead and security holes can be entirely avoided by using *natural* iterators. A natural iterator acts the other way around. That is, rather than repeatedly call an iterator (its `next()` method) to get hold of the next element of a collection, you merely need to specify what action is to be performed on the *current* element. You *do not* need to write any loop or check termination conditions: it is iterator's job. You can even maintain your own (lexical) local context during the iteration.

This technique - call an iterator and pass an iteratee - is widely used in "other" languages: Lisp, Scheme, ML, Prolog and Dylan. You do not even need to assign a name to an iteratee-function. It is somewhat surprising that the same thing is possible in C++, complete with anonymous functions and clauses, well, kind of.

Appendix 3 shows off a natural iteration in a snippet from my part editor. Here is another self-explanatory example:

```
void foo(IMAGE& Test_image) {
  struct SqrImage : public
PixelPrimAction {
   void operation(GRAY& pixel) {
 pixel = sqr((GRAY_SIGNED)pixel);
}
  };
  Test_image.apply(SqrImage());
}
```

If a C++ compiler handles templates well, then the entire iteration can be done inline:

```
ScanLines_Connector
scanlines_connectee(*this);
TwoScanLinesIterator
 <ScanLines_Connector>
   scanlines_connector(
  prev_scanline->get_scanline(),
  curr_scanline->get_scanline());
scanlines_connector.for_each(
            scanlines_connectee);
```

The `for_each` iteration above is actually inlined by the compiler, so it works just like a `for()`-loop, only safer and with less overhead: Since control never leaves the iterator until it finishes, the iterator does not have to check termination conditions several times.

One can point out two major kinds of iterators: `for-each` and `any?` (aka first-found). Each has a distinct programming pattern, which sometimes is not recognized. For example:

```
static void DrawNewSprite(void)
{
  register short slot;
  SpriteInfoPtr spriteInfoP;

  slot=gFirstSpriteSlot;
  while( slot >= 0 )
  {
    register short numRows;
    register short numCols;
    register short h;
    register short v;
    spriteInfoP =
        &gSpriteInfo[slot];
    if( spriteInfoP->status < 0 )
     goto nextSlot;
    numRows = spriteInfoP->height;
    numCols = spriteInfoP->width;
    h = spriteInfoP->position.h;
    v = spriteInfoP->position.v;
    if( h+ numCols <= 0 ||
        h >= gWindowWidth ||
        v+ numRows <= 0 ||
        v > qWindowHeight)
      goto nextSlot; /* totally
Offscreen */
    DrawSprite(slot);
  nextSlot:
    slot= spriteInfoP->nextSlot;
  }
}
```

Fig. 3a. Original snippet from "Programmer's Challenge," MacTech Magazine, September 1995, p.58

```
static void DrawNewSprite(const
int start_slot, const SpriteInfo
sprite_info [])
{
  register int slot;

  for(slot=start_slot; slot >= 0;
  slot=sprite_info[slot].nextSlot)
  {
    const SpriteInfoPtr
spriteInfoP = &sprite_info[slot];
    register int h,v;
    if( spriteInfoP->status < 0 )
      continue;
    h = spriteInfoP->position.h;
    v = spriteInfoP->position.v;
    if( h+ spriteInfoP->width > 0
        && h < gWindowWidth &&
        v+ spriteInfoP->height > 0
        && v < gWindowHeight)
      DrawSprite(slot);
  }
}
```

Fig. 3b. Fixed snippet explicitly recognizing for-each iteration

I can't help saying that goto in the original snippet, Fig. 3a, is *disgustingly* ***gross***. Obviously a goal of `DrawNewSprite` was to walk through a list and perform a certain regular action on each non-empty slot. This is a standard for-each-type iteration. Why would one program it with `goto`'s just beats me. Global variables are another bad thing in this code. It really takes *longer* to access global variables, especially on a PowerMac, where regular function arguments come already loaded into registers. Also, global variables lead to trouble with `A5/A4` (if one isn't careful), they make it difficult to put the code in a code resource or some sort of shared library (Component, plug-in, OpenDoc part). In this example, Fig. 3a, `gFirstSpriteSlot` and `gSpriteInfo` could have been function parameters. Fig. 3b shows a better, clearer and faster implementation of `DrawNewSprite` (keep in mind it is merely a C code, not C++).

Iterators go well along with object promises: here is an example of how to write "`Vector a = b+c;`" and not to worry about stack overflows, temporary objects and assignments.

```
class square_add : public
LazyMatrix, public ElementAction
{
  const Vector &v1;
  const Vector &v2;
  void operation(REAL& element)
    { assert(j==1); element =
     v1(i)*v1(i) + v2(i)*v2(i); }
```

```
  void fill_in(Matrix& m) const
    { m.apply(*this); }
  public: square_add(const
  Vector& _v1, const Vector& _v2)
  : LazyMatrix(_v1.q_row_lwb(),
             _v1.q_row_upb(),1,1),
    v1(_v1), v2(_v2) {}
};
Vector vres = square_add(v2,v3);
Vector vres1 = v2;
vres1 = square_add(v2,v3);
```

Here `square_add` promises to deliver a vector with elements being sums of squares of elements of two other vectors. The promise is forced either by a `Vector` constructor, or by assignment to another vector (in the latter case, a check is made that the dimensions of the promise are compatible with those of the target). In either case, computation of new vector elements is done "in place", no temporary storage is ever allocated/used. Iteration is also done "behind the scenes", relieving the user of worries about index range checking, etc.

What were they thinking?

Safe programming can be fun, too! Safe code *can* be cute, and fast. Let's consider a very telling example, from a public domain code to crack encoded resources in Win95:

```
unsigned char Data[100001];
main(...){
  int     size;
... opening a file
  size=0;
  while(!feof(fd)) {
    Data[size++]=fgetc(fd);
  }
  size--;
```

Q: what happens if the file turns out longer than 100001?

The *entire* snippet above is functionally equivalent to

```
size = fread(Data,1,
             sizeof(Data)-1,fd);
```

It is *much* faster, it is absolutely safe (there is not even a possibility of getting over `Data`'s boundary), and it is only <u>one line</u> for God's sake. The guy who wrote the code seems to be smart and knowledgeable. He should have written that one line on "autopilot". Come to think of it, there is little wonder then why M$ and other software companies goof it up all the time...

Of course it is hard to program safely when so many of the standard C functions are treacherous. A few have surpassed the others in notoriety: `gets()`, `strcpy()` and `strcat()`. `gets()` is directly responsible for all grand wormholes: the internet worm (fingerd), sendmail and httpd root holes. Even `gets()` manual page recommends against it:

```
char *gets(char *s);
When using gets(), if the length
of an input line exceeds the size
of s, indeterminate behavior may
result. For this reason, it is
strongly recommended that gets()
be avoided in favor of fgets().
```

Indeed, when allocating a buffer for `gets()` to read into, there is *really* no way of knowing how long the input line happens to be. What is worse, there is no way to prevent the overrun. This sets `gets()` aside as particularly insidious: at least in case of `strcpy()`, we can find out the size of the source string before copying. Still, `gets()` proliferates. Is it really so hard to replace it with `fgets()`, like in a snippet below:

```
char Work_file_name[50];
fprintf(stderr,
        "\nWork file name >");
fgets(Work_file_name,
   sizeof(Work_file_name),stdin);
Work_file_name[
   sizeof(Work_file_name)-1] =
     '\n';
*strchr(Work_file_name,'\n') =
     '\0';
```

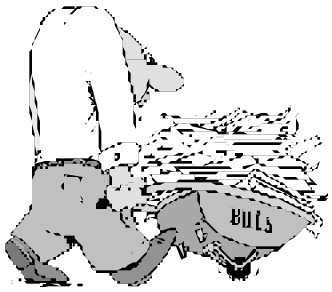
This works just as the original `gets()`, but <u>never</u> writes beyond buffer's boundaries. Unexpectedly long input is simply truncated. An enhanced snippet in Appendix 4 can also detect `EOF`, system and user input errors.

Here is another telling example, from an article "Learning C++" in January 1996 issue of "Software Development"

```
    //static character buffer
template <class Len> class
CHARBUF
{
public:
  CHARBUF(const char * str = 0)
```

```
    { strlen(str) > strlen(buf) ?
       abort() : strcpy(buf,str); }
    operator char*()
       { return buf; }
    char buf[LEN];
};
```

Disregard for a moment a flawed design: it will be interesting to see the result of `CHARBUF<complex> buf;` Definitely the author meant `<int LEN>` rather than `<class LEN>` in the template, as only integers are allowed to index an array of characters. But there are more serious faults: for one thing, array `buf` is not initialized. Therefore, `strlen(buf)` in the constructor can return anything, including 0 (in which case the constructor aborts). The main gripe though is that `str` defaults to 0 in the constructor, which is never checked. That is, `strlen()` and then `strcpy()` in the constructor could receive a zero pointer. Ironically, a subtitle of that article reads: "A simple, object-oriented technique called 'brief classes' can make C++ a much safer *(sic!)* language in which to work." They call it safer nowadays...



Appendix 1. Example of late binding of incorporated objects

```
class VenusDialog : public ModelessDialog
{
  enum { item_Go = 1, item_Cancel = 2, item_2dview = 3,
         item_3dview = 4,  item_ze=5,   item_ze_val=6,
         item_ze_text=7 };
  enum { dlog_id = 128 };

  ProjectionParameters projection_parms;
  Boolean handle_item_hit(const int item_no);
  ValueControl eyepoint_elevation;
   // snipped
  ImageView image_2d_view;
  ThreeDView image_3d_view;
  void draw_user_item(const int item_no);

  Boolean is_flying; // Is our plane moving?
public:
  VenusDialog(const IMAGE& image);
};
```

All dialog items need a pointer to a completely initialized dialog; unfortunately, the constructor for the items is called *before* the ctor for the whole object. Still, there is a hope, and there is a way:

```
VenusDialog::VenusDialog(const IMAGE& image)
   : ModelessDialog(dlog_id), image_2d_view(image),
     image_3d_view(image,image_2d_view.where_is_viewer(),
                   projection_parms),
     is_flying(FALSE)
```

```
{
                    // Late construction of items, after the dialog itself has been initialized
  image_2d_view.bind(*this,item_2dview);
  eyepoint_elevation.bind(ControlItem(*this,item_ze),
                          TextItem(*this,item_ze_val));
  projection_parms.ze = eyepoint_elevation;

  image_3d_view.bind(*this,item_3dview);
  show();
}
```

Appendix 2. Precision targeted heap pointers

```
            // Generic dir item being added to the directory
            // Note the item (and specialized items derived from it)  do not have
            // a public constructor;  calling function New() is the only  way to
            // build an item. The "pointer"  the function returns is supposed to
            // be += to the directory: that's the only  thing the "pointer" is good for
class TIFFNewDirItem : public TIFFDirEntry
{
  friend class TIFFBeingMadeDirectory;

  TIFFNewDirItem * next;

protected:

  class ItemRef       // This is a wrapper for TIFFNewDirItem*
  {                   // It guarantees that the pointer
    friend class TIFFBeingMadeDirectory;      // to the object is really
    TIFFNewDirItem * const ref;               // inaccessible to everyone
    TIFFNewDirItem& surrender(void)           // but the directory
        { return * ref; }
    //  { const TIFFNewDirItem& item = ref; ref = 0; return item; }
  public:
    ItemRef(TIFFNewDirItem * item) : ref(item) {}
    //~ItemRef(void) { if( ref ) delete ref; }
  };
  virtual void write(EndianOut& file) = 0;    // Write this field into a file
  virtual void write_value(EndianOut& file) = 0;// Write additional data
  TIFFNewDirItem(const short _tag, const DataType _type,
                 const long _count, const long _value)
        : TIFFDirEntry(_tag,_type,_count,_value), next(0) {}
 };

          // Dictionary under construction.  Inserted items automatically arranged
          // in the ascending order of their tags
class TIFFBeingMadeDirectory
{
  TIFFHeader header;
  card no_entries;
  TIFFNewDirItem * first_entry;   // Other items are chained to that

public:
  TIFFBeingMadeDirectory(void);
  void operator += (const TIFFNewDirItem::ItemRef& ref);
```

```
  ~TIFFBeingMadeDirectory(void);
 void write(EndianOut& file);           // Write the entire TIFF directory
};

          // That is how this all is used
void IMAGE::write_tiff(const char * file_name,const char * title,
                       const TIFFUserAction& user_adding_tags) const
{
  is_valid();

  message("\nPreparing a TIFF file with name '%s'\n",file_name);

  EndianOut file(file_name);
  TIFFBeingMadeDirectory directory;

  directory += ScalarTIFFDE::New(TIFFTAG_IMAGEWIDTH,(unsigned)q_ncols());
  directory += ScalarTIFFDE::New(TIFFTAG_IMAGELENGTH,(unsigned)q_nrows());
  directory += ScalarTIFFDE::New(TIFFTAG_COMPRESSION,
                                 (unsigned short)COMPRESSION_NONE);
  directory += RationalTIFFDE::New(TIFFTAG_XRESOLUTION,72,1);
  if( name != 0 && name[0] != '\0' )
    directory += StringTIFFDE::New(TIFFTAG_IMAGEDESCRIPTION, name );
  user_adding_tags(directory);          // Give the user a chance to add
                                        // his own tags
  directory.write(file);
  file.close();
}
```

Appendix 3. Natural iteration in an OpenDoc part

```
class selection_checker : public ShapeIteratee
{
  CSelection * selection;
  ODBoolean was_selected_flag;
  ODBoolean operation(CShape * shape)
    { return !(was_selected_flag = selection->IsIn(shape)); }
public:
  selection_checker(CSelection * _selection)
    : selection(_selection), was_selected_flag(kODFalse) {}
  operator ODBoolean (void) const
    { return was_selected_flag; }
};
selection_checker was_selected(fSelection);
for_each_our_shape(was_selected);

class selection_adder : public ShapeIteratee
{
  Environment * ev;
  CSelection * selection;
  ODBoolean operation(CShape * shape)
    { if( !(selection->IsIn(shape)) )
        selection->AddToSelection(ev, shape, kODFalse);
      return kODTrue; }
public:
  selection_adder(Environment * _ev, CSelection * _selection)
```

```
    : ev(_ev), selection(_selection) {}
};
if( was_selected )
  for_each_our_shape(selection_adder(ev,fSelection));
```

Appendix 4. Safe reading of a string from a file

```
            // Read a string from a file: a very safe function
            // Returns a ptr to the STATIC array;  Returns NULL on eof
const char * get_string(FILE * fp)
{
  static char buffer[120];
                  // if fgets() returns 0 => it failed
  if( fgets(buffer,sizeof(buffer)-2,fp) == (char *)0 )
    if( feof(fp) )
      return (char *)0;
  else
    perror("Reading error"), exit(4);

                  // If we read an entire string, the buffer should contain \n\0 at
                  // the very end. If there is no '\n', it means either we got EOF
                  // or the string was too big to fit into the buffer
  if( buffer[strlen(buffer)-1] != '\n' )
    if( feof(fp) )
      return buffer;
    else
      _error("The string <%s> is way too big. Too bad!",buffer);

  buffer[strlen(buffer)-1] = '\0';      // Replace '\n' with '\0'

  return buffer;
}
```