Code and Personality

How to tell your personality type from your code.

Kevin Marks <kmarks@apple.com>
Maf Vosburgh <maf@goat.demon.co.uk>

People with different personalities write the same code different ways. Unfortunately, not all these ways actually work, but people can learn to change…

Introduction

There are various types of programmers around. We've certainly worked with a wide selection. Over the years, we've come to realize that programmers can be divided into various "personality types". You don't stay the same personality-type your whole life though — as you develop and learn, your approach to programming changes and that change is visible in your code. We're going to look at various functions and how programmers with different personalities would write them.

MacHack attendees have normally been around the block a few times. That means they have learnt various things, like when you're going around the block, it helps to watch where you're going, and be driving a tank. We know that a function has important responsibilities. It needs to check every error code, keep track of every byte it allocates, and that function needs to know how to cope with anything that happens, cleaning up perfectly after itself and returning an error code which explains what went wrong. But in order to write code like this you have to have made mistakes and learned from them. We know we have.

Optimistic

Everyone starts out by being Optimistic. Optimistic programmers assume that system calls will always succeed, there is always enough memory and disc space, and there really is a Santa Claus.

You can spot Optimistic programmers by their novel approach to error checking. For instance, here's an Optimistic function to return a memory buffer filled with 0xFF.

```
char * GetTheBuffer(
  unsigned long  bufferSize)
{
  char    *p = NewPtr(bufferSize);
  unsigned long  x;

  for (x = 0  ; x < bufferSize ; x++)
    p[x] = 0xFF;

  return p;
}
```

Listing 1: *Optimists get to see a lot of Macsbug.*

Guess what happens if NewPtr fails? Yes, it returns NULL and you then overwrite say the low ten megs of memory, depending on the value of bufferSize. Cool.

Petulant

You can't stay an Optimist once you've watched your code crash horribly a few dozen times. For many, the next step is to become Petulant.

```
char * GetTheBuffer(
  unsigned long  bufferSize)
{
  char    *p = NewPtr(bufferSize);
  unsigned long  x;

  if (p == NULL)
    DebugStr(
      "\p Error- cannot allocate buffer!");

  for (x = 0  ; x < bufferSize ; x++)
    p[x] = 0xFF;

  return p;
}
```

Listing 2. *Petulant programmer going critical..*

As you can see, Petulant programmers detect that something has gone wrong, complain about it in a primitive way, and then they crash your machine anyway. Not very useful. Any error you detect, you should cope with.

Stoic

The next stage in evolution is to handle any errors that occur, but keep quiet about them. This is the sort of "stiff upper lip" approach that built the British Empire (and made sure it didn't last). As Pink Floyd said on Dark Side of the Moon, "Hanging on in quiet desperation is the English way". Let's call this personality Stoic.

```
char * GetTheBuffer(
  unsigned long  bufferSize)
{
  char     *p = NewPtr(bufferSize);
  unsigned long  x;

  if(p != NULL)
    {
    for (x = 0  ; x < bufferSize ; x++)
      p[x] = 0xFF;
    }

  return p;
}
```

Listing 3. *Stoics suffer in silence.*

This approach is pretty good - if the allocation fails the routine does not trash memory, and it returns NULL, which is a pretty standard thing to do when an allocation fails. The trouble with this style, is that you are not telling the caller why you failed. Some functions can fail for any number of different reasons, and it helps to pass that information on. Furthermore, you are assuming that the caller realizes that your function will sometimes return NULL. If the person calling your function is an Optimist, you're back at square one, as they will happily take that NULL buffer and start using it.

Realists

Of course grown-up programmers (the sort who come to MacHack) are Realists. They know that problems are inevitable, and when things go wrong, you need to tell the caller about that. Also, if your function returns OSErr, just by looking at the prototype they can see that errors may happen, and know that they need to plan for that.

```
OSErr GetTheBuffer(
  unsigned long bufferSize, char *returned)
{
  OSErr result = noErr;
  char*p =  NewPtr(bufferSize);
  unsigned long  x;

  if (p == NULL)
    result = memFullErr;
  else
    {
    for (x = 0  ; x < bufferSize ; x++)
      p[x] = 0xFF;
    }

  *returned = p;
  return result;
}
```

Listing 4. *Realists do the right thing.*

You could just stop here - at this point the code is doing the right thing, or telling the caller why it couldn't. It isn't leaking or trashing memory.

Considerate

However you may want to help others see the errors of their ways (in the debug build at least), and become a Considerate programmer.

```
OSErr GetTheBuffer(
  unsigned long  bufferSize,
  char *returned)
{
  OSErr result = noErr;
  char*p =  NULL;
  unsigned long  x;

  if (bufferSize == 0)
    {
    DebugText("GetTheBuffer:
      a zero-size buffer is no use");
    result = paramErr;
    }
  else
```

```
    {
    p = NewPtr(bufferSize);
    if (p == NULL)
      {
      DebugText(
"GetTheBuffer: couldn't get the buffer");
      result = memFullErr;
      }
    else
      {
      for (x = 0  ; x < bufferSize ; x++)
        p[x] = 0xFF;
      }
    }
  *returned = p;
  return result;
}
```

Listing 5. *Considerate programmers keep you informed.*

DebugText is defined something like this:

```
#ifdef DEBUG
#define DebugText(x) debugstr(x)
#else
#define DebugText(x)
#endif
```

DEBUG is the preprocessor global you use to distinguish between the debugging and final builds of your app. You could define DebugText to log to a file, or put up an alert - the point is to have a way of signaling to other programmers (or yourself) that something odd is going on in your routine, but to protect the end user from it by coping nicely.

The programmer here not only returns sensible error values, but in the debug build he also stops the program when passed a buffer that is too big - handy for debugging the calling code too. However, by not using an assert, he doesn't inflict this on the end-user. That would be not just Petulant, but Sociopathic as well.

Extravagant

A common personality type that you come across is the Extravagant programmer. Extravagant programmers back out when an error happens, but they don't bother disposing of the stuff they allocated along the way. That causes programs that work fine until an error occurs, causing a leak, which takes up memory and causes another error, and so on until the program eventually fails catastrophically.

The trend to use C++ exceptions contributes to this problem, since they make it so easy to just jump up several levels of execution, happily leaking all sorts of things

ResizeGrayScalePicHandle is a C example of a function that needs to allocate a bunch of things and do some real work. **Some little things we need to define before the example:**

```
const short k8BitGrayScaleID = 32 + 8;

void QDNormal(void)
{
  ForeColor(blackColor);
  BackColor(whiteColor);
  PenNormal();
}

// make the left and top coords 0
void ZeroRect(Rect *r)
{
  OffsetRect(r, -r->left, -r->top);
}


// zero the position and scale
void ScaleRect(Rect  *r, double scale)
{
  ZeroRect(r);
  r->right = ((double)r->right) * scale;
  r->bottom = ((double)r->bottom) * scale;
}

OSErr ResizeGrayScalePicHandle(
  PicHandle inPic, PicHandle *outPic,
  double  scale)
{
  Rect  inRect = inPic [0]-> picFrame[1];
  Rect  outRect = inRect;
  GWorldPtr inWorld, outWorld;
  GDHandle saveGD;
  CGrafPtr savePort;
  CTabHandle grayTab =
    GetCTable(k8BitGrayScaleID);
```

*1 The sequence [0]-> is called "sprong", and is a convenient way to use a typed Handle to a struct. We could have used "inRect = (**inPic).picFrame" with the same result.* ***Sprong is particularly useful when dealing with chains of typed Handles to structs, as in this example:***
short depth = GetMainDevice() [0]-> gdPMap [0]-> pixelSize;

```
  if (grayTab == NULL)
    return memFullErr;

  ZeroRect(&inRect);
  ScaleRect(&outRect, scale);
  result = NewGWorld(&inWorld, 8, &inRect,
    grayTab, NULL, 0);
  if (result != noErr)
    return result;

  GetGWorld(&savePort, &saveGD);
  SetGWorld(inWorld, NULL);

  result = NewGWorld(&outWorld, 8,
    &outRect, grayTab, NULL, 0);
  if (result != noErr)
    return result;

  LockPixels(GetGWorldPixMap(inWorld));
  LockPixels(GetGWorldPixMap(outWorld));
  EraseRect(&inRect);
  DrawPicture(inPic, &inRect);

  SetGWorld(outWorld, NULL);
  CopyBits( (BitMap*) *inWorld->portPixMap,
    (BitMap*) *outWorld->portPixMap,
    &inRect, &outRect, ditherCopy, NULL );
  *outPic = OpenPicture(&outRect);
  CopyBits( &qd.thePort->portBits,
    &qd.thePort->portBits, &outRect,
    &outRect,srcCopy, NULL);
  ClosePicture();

  DisposeGWorld(inWorld);
  DisposeGWorld(outWorld);
  DisposeCTable(grayTab);
  SetGWorld(savePort, saveGD);
  return result;
}
```

Listing 6. *Extravagant programmer wants your heap.*

You can see that there is an attempt at error handling, but if the program fails it leaves things in a sorry state. If a failure occurs it can leak a GWorld, a color table, and can also leave the QuickDraw port set to one of the leaked offscreen GWorlds.

This kind of real-world function shows how complicated real error handling can be, but luckily there is a simple style that can cope with this sort of problem. Just keep all the cleaning up code at the end of the function, as above, and write it in such a way that it always safe to execute, and just jump to that block if an error occurs.

```
OSErr ResizeGrayScalePicHandle(
  PicHandle inPic, PicHandle *outPic,
  double  scale)
{
  Rect  inRect = inPic [0]-> picFrame;
  Rect  outRect = inRect;
  GWorldPtr inWorld = NULL,
    outWorld = NULL;
  GDHandle saveGD;
  CGrafPtr savePort;
  CTabHandle grayTab =
    GetCTable(k8BitGrayScaleID);

  GetGWorld(&savePort, &saveGD);

  if (grayTab == NULL)
    {
    result = memFullErr;
    goto bail;
    }

  ZeroRect(&inRect);
  ScaleRect(&outRect, scale);

  result = NewGWorld(&inWorld, 8, &inRect,
    grayTab, NULL, 0);
  if (result != noErr)
    goto bail;

  result = NewGWorld(&outWorld, 8,
    &outRect, grayTab, NULL, 0);
  if (result != noErr)
    goto bail;

  SetGWorld(inWorld, NULL);
  LockPixels(GetGWorldPixMap(inWorld));
  LockPixels(GetGWorldPixMap(outWorld));
  QDNormal();
  EraseRect(&inRect);
  DrawPicture(inPic, &inRect);
  SetGWorld(outWorld, NULL);
  QDNormal();
  CopyBits( (BitMap*)*inWorld->portPixMap,
    (BitMap*) *outWorld->portPixMap,
    &inRect, &outRect, ditherCopy, NULL );
  DisposeGWorld(inWorld);
  inWorld = NULL;

  *outPic = OpenPicture(&outRect);
  CopyBits( &qd.thePort->portBits,
    &qd.thePort->portBits,
    &outRect, &outRect,srcCopy, NULL);
  ClosePicture();

bail:
  if (inWorld)
    DisposeGWorld(inWorld);
  if (outWorld)
    DisposeGWorld(outWorld);
  if (grayTab)
    DisposeCTable(grayTab);
```

```
  SetGWorld(savePort, saveGD);
  return result;
}
```

Listing 7. *Cleans up properly on error now, but a little verbose.*

Notice how everything is always disposed, there is no duplication of cleanup code, and all errors are checked and returned. Initialising all storage to NULL, means that the cleanup code can easily see what needs to be disposed. See how when we dispose inWorld in the middle of the function, we set it to NULL, so the cleanup code knows not to dispose it again.

Of course some people say that goto is evil, and just typing those four letters is enough to condemn you to Silicon Hell. We disagree - this particular use of goto is an elegant solution to a particularly nasty problem. It removes the need to need to duplicate the cleanup code or get into a confusing case of nesting.

This style gets more compact with the use of the BailOSErr macro, first seen by us in QuickTime sample code.

```
#define BailOSErr(a) {result = (a) ;
  if (result != noErr) goto bail;}[2]
```

By using the BailOSErr macro a block like this:

```
result = NewGWorld(&inWorld, 8, &inRect,
  grayTab, NULL, 0);
if (result != noErr)
  goto bail;
```

or this:

```
if (result = NewGWorld(&inWorld, 8,
&inRect, grayTab, NULL, 0))
  goto bail;
```

turns into this line :

```
BailOSErr( NewGWorld(&inWorld, 8, &inRect,
  grayTab, NULL, 0) );
```

It is also helpful to write dispose functions for commonly disposed toolbox objects like the two below. These ignore NULL values, and write NULL back to the disposed variable, preventing a double dispose.

```
void DisposeIfGWorld(GWorldPtr  *gw)
{
  if ( (gw != NULL) && (*gw != NULL))
    {
    DisposeGWorld(*gw);
    *gw = NULL;
    }
}

void DisposeIfCTable(CTabHandle *ct)
{
  if ( (ct != NULL) && (*ct != NULL))
    {
    DisposeCTable(*ct);
    *ct = NULL;
    }
}
```

Listing 8. *Wrapper functions for disposing Toolbox objects.*

Here is a version using the new stuff:

```
OSErr ResizeGrayScalePicHandle(
  PicHandle inPic, PicHandle *outPic,
  double scale)
{
  Rect  inRect = inPic[0]->picFrame,
        outRect = inRect;
  GWorldPtr  inWorld = NULL,
             outWorld = NULL;
  GDHandle saveGD;
  CGrafPtr savePort;
  CTabHandle grayTab =
    GetCTable(k8BitGrayScaleID);

  GetGWorld(&savePort, &saveGD);

  if (grayTab == NULL)
    BailOSErr(memFullErr);
```

2 Actually, this macro would be better written wrapped with "do/while", like this
#define BailOSErr(a) do {result = (a) ; if (result != noErr) goto bail;} while(0)
because then you can write "if (x) BailOSErr(foo()); else BailOSErr(bar());" without getting an error message from the C compiler.

```
  ZeroRect(&inRect);
  ScaleRect(&outRect, scale);
  BailOSErr( NewGWorld(&inWorld, 8,
    &inRect, grayTab, NULL, 0));
  BailOSErr( NewGWorld(&outWorld, 8,
    &outRect, grayTab, NULL, 0));
  SetGWorld(inWorld, NULL);
  LockPixels(GetGWorldPixMap(inWorld));
  LockPixels(GetGWorldPixMap(outWorld));

  EraseRect(&inRect);
  DrawPicture(inPic, &inRect);

  SetGWorld(outWorld, NULL);
  CopyBits( (BitMap*)*inWorld->portPixMap,
    (BitMap*) *outWorld->portPixMap,
    &inRect, &outRect, ditherCopy, NULL );

  DisposeIfGWorld(&inWorld);

  *outPic = OpenPicture(&outRect);
  CopyBits( &qd.thePort->portBits,
    &qd.thePort->portBits, &outRect,
    &outRect,srcCopy, NULL);
  ClosePicture();

bail:
  DisposeIfGWorld(&inWorld);
  DisposeIfGWorld(&outWorld);
  DisposeIfCTable(&grayTab);

  SetGWorld(savePort, saveGD);
  return result;
}
```

Listing 9. *Macros & dispose wrappers, make Realism compact.*

The step beyond Realism is to add a dose of Cynicism. We like to tag all our data structures, so we can recognise them when they are passed in again. The overhead is low, and it makes your modules almost bullet-proof.

We write in a component orientated style, based on the way the Mac Toolbox behaves. Programs we've written, like "3D Atlas", use all sorts of objects. For instance we've written our own picture buttons, which draw in various sophisticated ways. To create one of these you call NewButton which gives you a ButtonHandle. To draw it you pass that ButtonHandle to DrawButton, to dispose it you call DisposeButton, etc.

Structures like ButtonHandles have a tag at the front. This is an OSType - in the case of a ButtonHandle it is initialised to the value 'Bttn' when a ButtonHandle is created.

```
// some fields removed for clarity
typedef struct
  {
  OSType  tag;  // here is the tag
  Rect    buttonRect;
  short   baseID;
  }ButtonRec, *ButtonPtr, **ButtonHandle;
```

Listing 10. *Tag it, so you'll recognise it again.*

Cynical

These tags are fabulously useful things, if like us you have gone beyond being Realists and have become really quite Cynical. All externally visible functions in Button.c verify the ButtonHandle argument they were passed is not NULL, is not a purged handle, and that it has the correct tag. This is done with a macro called "Check" that is defined at the top of Buttons.c.

```
static OSType kTag  = 'Bttn';
static char kTypeName[] = "ButtonHandle";

#define Check(b) CheckObject((Handle)b,
  kTag, kTypeName)
```

CheckObject is a function that is used throughout the program:

```
OSErr CheckObject(Handle h, OSType tag,
  char *expectedType)
{
  OSErr   result = noErr;

  if (h == nil)
    result = nilHandleErr;
  else if (*h == nil)
    result = nilHandleErr;
  else if (*(long*)*h != tag)
    result = paramErr;

#ifdef DEBUG
  if (result)
    {
    charstr[256];
```

```
    sprintf( str,
      "CheckObject failure type %hd on %s",
      result, expectedType);
    DebugText(str);
    }
#endif

  return result;
}
```

Listing 11. *CheckObject*

Here's a simple example which uses the Check macro.

```
OSErr ShowButton(ButtonHandle  button)
{
  OSErr   result  = Check(button);

  if (result == noErr)
    {
    button [0]-> visible = true;
    result = DrawButton(button);
    }

  return result;
}
```

Listing 12. *It's easy to check that Handles are the right type.*

If you call ShowButton and instead of passing it a ButtonHandle you pass it NULL, a purged Handle, or a PicHandle containing a picture of your grandmother, that error will be detected immediately. We've found that this is mainly useful for catching coding errors while still in the development phase, and that the most common error is to try and act on a NULL object.

The tag also enables you to recognise ButtonHandles when browsing through the heap with a tool like ZoneRanger.
Lastly, the DisposeButton routine sets the tag to 0, so it is not possible to dispose a valid ButtonHandle twice.

```
void DisposeButton(ButtonHandle theButton)
{
  if (Check(theButton) == noErr)
    {
    theButton [0]-> tag = 0;
    // dispose any storage owned by the object here
    DisposeHandle((Handle)theButton);
    }
}
```

Listing 13. *Clear the tag before you dispose.*

For the cost of some little 4-byte tag, your program's internal routines can gain a lot more ruggedness.

Those Damn Optimists Again

The Mac Resource Manager is a bit of a mine-field for the Optimist.

For instance, what's wrong with this code that uses purgeable 'PICT' resources?

```
OSErr DrawTwoPictures(short id1, short id2)
{
  Rectr = {0,0,100,100};
  Handlepic1 = GetResource('PICT', id1);
  Handle  pic2 = GetResource('PICT', id2);

  DrawPicture(&r, (PicHandle)pic1);
  DrawPicture(&r, (PicHandle)pic2);

  return noErr;
}
```

Listing 14. *Might work, might not,*

The previous function assumes a lot of things. It assumes that the resources both exist, can fit in memory together, have not been purged already, and don't get purged in the course of the function. When one of those assumptions fails, you will be in trouble.

Purgeable resources are a very powerful tool, giving your app a free caching scheme, but a lot of people forget that GetResource can give you back a purged Handle, or that loading one resource can push out another.

To challenge some of these assumptions it helps to have a function like this to call. If CheckResource says the resource Handle is OK, and you then make it non-purgeable, you can count on using it in your code.

```
// verify a resource handle, reloading it if needed
OSErr CheckResource(Handle     h)
{
  if (h == NULL)
    return resNotFound;
  if (*h == NULL)
    {
    LoadResource(h);
    if (*h == NULL)
      return memFullErr;
    }
  return noErr;
}
```

Listing 15. *CheckResource is your friend.*

Realism In Action

Given the CheckResource function, it is possible to write a more Realist version of this function with blocks like this:

```
pic1 = GetResource('PICT', id1);
if (result = CheckResource(pic1))
  {
  pic1 = NULL;
  goto bail;
  }
pic1State = GetHState(pic1);
HNoPurge(pic1);
```

Listing 16. *Real programmers save and restore state.*

You restore the Handle's (probably purgeable) state at the end with HSetState.

This code can be made a lot less cumbersome the use of a simple function called SafeGetResource. This function gets the valid resource data, saving the Handle's state (locked, purgeable, etc), and makes it non-purgeable. To clean up at the end, just use HSetState to return the Handle to its initial state (ie purgeable).

```
OSErr SafeGetResource( Handle *retHandle,
  char *retState, ResType theType,
  short theID)
{
  Handleh = GetResource(theType, theID);
  OSErr   result = noErr;

  if (result = CheckResource(h))
    {
    *retHandle = 0;
    // only complain on the debug build
    DebugText("SafeGetResource failed");
    }
  else
    {
    *retState = HGetState(h);
    *retHandle = h;

    HNoPurge(h);
    }

  return result;
}
```

Listing 17. *Making your resource code bullet-proof.*

Why is it important to make a purgeable resource non-purgeable as soon as you've made sure it is loaded?

Because it is not possible to count on a purgeable resource after you have made any calls that could have caused the heap to be purged, and that means any calls that allocate memory, directly or indirectly.

This following code is bogus, for instance, because the second resource can push the first out of memory when it loads. You can only count on resources in non-purgeable Handles.

```
h1 = GetResource('PICT', 1);
BailOSErr( CheckResource(h1) );

h2 = GetResource('PICT', 2);
BailOSErr( CheckResource(h2) );
// by this point we have the second PICT, but that PICT could
// have pushed the first out of memory
```

Now we have enough tools to write a version of the function that cannot be made to misbehave by missing resources, limited memory or purged Handles. Note that we do not use ReleaseResource in this case since we know these resources were originally marked purgeable, so we'll get the memory eventually if we need it.

```
OSErr DrawTwoPictures(short id1, short id2)
{
  Handlepic1 = NULL;
  Handlepic2 = NULL;
  Rect  r = {0,0,100,100};
  char  pic1State, pic2State;
  OSErr   result = noErr;
```

```
  BailOSErr( SafeGetResource( &pic1,
    &pic1State, 'PICT', id1) );
  BailOSErr( SafeGetResource( &pic2,
    &pic2State, 'PICT', id2) );

  DrawPicture(&r, pic1);
  DrawPicture(&r, pic2);

bail:
  if (pic1)
    HSetState(pic1, pic1State);

  if (pic2)
    HSetState(pic2, pic2State);

  return result;
}
```

Listing 18. *With SafeGetResource, your purgeable resource code is robust.*

Myopic

Some programmers don't see what their code does very clearly, so we call them "myopic" (short-sighted). We had a job applicant once who proudly showed us his code, including his own version of strcpy. He claimed the version in the standard library didn't work (which seemed unlikely). We were immediately suspicious, so we checked out his version.

```
void strcpy(char *out, char *in)
{
  int x;

  for (x = 0 ; x <= strlen(in)  ; x++)
    out[x] = in[x];
}
```

Listing 19. *Slow version of strcpy.*

We don't think the programmer meant to measure the length of the source string again, every single time around the loop. Of course, fixing that bug would not be enough in itself, since the routine has other faults too.[3]

Obfuscators

A related character flaw is the person who optimises the amount of on-screen space the code takes up, instead of the compiled output. We would call these Obfuscators.

This routine is supposed to make all the characters in a string lower case.

```
voidLowText(char *a)
{
  char*b=a;
  while(*a) *b++ =
    (islower(*a))? *a++:*a++ + ('a'-'A');
  *b = 0;
}
```

Listing 20. *Obfuscator at work.*

We're not sure why writing code like this is popular, but it certainly isn't clear. Let's reformat it to be less secretive and obfuscated:

```
static const char kLowerCaseOffset =
  'a'-'A';

voidLowText(char* inStr)
{
  char* outStr = inStr;

  while(*inStr != 0)
    {
    if (islower(*inStr))
      *outStr++ = *inStr++;
    else
      *outStr++ = *inStr++ +
        kLowerCaseOffset;
    }

  *outStr = 0;
}
```

Listing 21. *Less Obfuscated, so it's easier to see the bug.*

Now we can see what it's doing, it is a lot more obvious that it will not cope with any input text that isn't letters - for example it will turn '0' to '9' into 'P' to 'Y'.

It's also Prejudiced - it assumes the text is in English, and the ASCII character set, so the Japanese version isn't going to work very well. It also scorns the rather nice Toolbox routines that deal with text, which could be called Arrogant. To be Tolerant, you could rewrite it like this:

3 Other faults - it returns void instead of a pointer to the destination string, only copies strings up to a certain length, and is not very fast. A valid strcpy can be written very compactly in C though, if you try.

```
OSErr LowText(char *inStr)
{
  if (inStr == nil)
    {
    DebugText(
      "nil string passed to LowText");
    return paramErr;
    }

  LowercaseText(inStr, strlen(inStr),
    smSystemScript);
  return noErr;
}
```

Listing 22. *In this case it's best to let the OS do it anyway.*

Another way of being Secretive and Obfuscatory is to use C++ overloading in conjunction with inheritance between lots of classes so that the only way of telling which function is being called is with a source-level debugger. We'd give example code here, but it would take about six source files to demonstrate properly...

Paranoid

Paranoid programmers worry about things that can't possibly go wrong. They end up creating lots of extra work for themselves, because they are always looking in the wrong place for trouble.

Take a simple function like this:

```
Boolean RectIsOnMainScreen(Rect r)
{
  Rect  resultRect;
  GDHandle gd = GetMainDevice();

  return SectRect(&r,
    &gd [0]-> gdRect, &resultRect);
}
```

Listing 23. *Simple function.*

A Paranoid programmer would write it like this:

```
Boolean RectIsOnMainScreen(Rect r)
{
  Rect  resultRect;
  GDHandle gd = GetMainDevice();
  Boolean doesIntersect;
  char  hState;

  hState = HGetState( (Handle) gd );
  HLock((Handle)gd); // totally time
                     // consuming and bogus
  doesIntersect = SectRect(&r,
    &gd [0]-> gdRect, &resultRect);
  HSetState((Handle)gd, hState);
  return doesIntersect;
}
```

Listing 24. *Paranoid version.*

All the above code that messes around with the locked state of the Handle is completely un-needed. SectRect does not move memory. We have even seen a more pathological variant of this – where a programmer calls HLockHi for no good reason, causing heap compactions all over the place. Even if SectRect was a function that moved memory, using a temporary Rect variable would be a better solution.

But the consequences of Paranoia can be much darker, especially when combined with Prejudice. The Mercutio bug that caused panic some months ago was caused by some code similar to this:

```
Boolean RectIsOnMainScreen(Rect r)
{
  Rect  resultRect;
  GDHandle gd = GetMainDevice();
  Boolean doesIntersect;

  HLock((Handle)gd);
  doesIntersect = SectRect(&r,
    &gd [0]-> gdRect, &resultRect);
  HUnlock((Handle)gd, hState);
  return doesIntersect;
}
```

Listing 25. *Paranoia + Prejudice = Trouble.*

The above code doesn't bother to save and restore the state of the GDHandle, and always leaves it unlocked. Since GDHandles should ALWAYS be locked (unless pain is something that you enjoy), this is a Bad Thing.

Writing a version of SectRect that does move memory is left as an exercise for the reader.

Conclusion

We're optimistic about cynicism.

Cynicism works well as a programming personality because it leads you to expect the worst, so you are forced to write code that can cope with that situation.

And the truth is that failure is far more likely than a lot of programmers seem to think, given that it can be caused by actions including :

- Allocating even one byte of storage,
- Reading from the disc.
- Writing to the disc.
- Using a purgeable block of memory.
- Using a parameter that may be invalid.
- Calling another function that does one of the above.

It has been pointed out before that one good thing about being a Cynic, is that you are so rarely disappointed.

Bibliography

Further reading: TN 1118, TN 1120,
"Code Complete" by Steve McConnell,
Microsoft Press,
"Writing Solid Code" by Steve Maguire,
Microsoft Press

About the authors

Kevin Marks works for Apple Computer in the QuickTime Group.

Maf Vosburgh is a freelance engineer, last seen interviewing in the Bay Area.

By an amazing coincidence, both are British, both like sushi, and both used to work at the same company in London, where they both wrote "3D Atlas" (well, half each).