

# Appendix A: Code Listings [Writing Bug-Free C Code](#)

[A.1 APP.H Include File](#)

[A.2 BOOK.H Include File](#)

[A.3 HEAP.C Module](#)

[A.4 RAND.C Module](#)

[A.5 DOS.C Module](#)

[A.6 OutputDebugString\(\) for MS-DOS Programmers](#)

[A.7 ReportWinAssert\(\)](#)

This appendix brings together in one location all the code that was presented in this book. Also available online in a ZIP at

<http://www.duckware.com/bugfreec/source.zip>

## A.1 APP.H Include File

APP.H is a place holder include file that represents the global include file for your application. This is the one include file for all your modules. As you write new modules, you will need to modify this include file.

```
APP.H
/*--- Standard Includes -----
*/
#include <string.h>           /* strcpy/memset
*/
#include <stdio.h>            /* sprintf
*/
#include <stdlib.h>           /* malloc/free
*/

/*--- The include file for this book -----
*/
#include "book.h"

/*--- NEWHANDLE section -----
*/
NEWHANDLE (HRAND);
NEWHANDLE (HDOSFH);

/*--- USE_* section -----
*/

#ifdef USE_HRAND
```

```

/*-----
-
*
*   Random number generator
*
*-----
*/

EXTERNC HRAND APIENTRY RandCreate  ( int );
EXTERNC HRAND APIENTRY RandDestroy ( HRAND );
EXTERNC int   APIENTRY RandNext    ( HRAND );
#endif

#ifdef USE_LOWIO
/*-----
*
*   Access to low-level I/O run-time library functions
*
*-----*/

#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#endif

#ifdef USE_HDOSFH
/*-----
*
*   Code wrapper to low-level file I/O
*
*-----*/

EXTERNC HDOSFH APIENTRY DosOpenFile  ( LPSTR );
EXTERNC WORD   APIENTRY DosRead      ( HDOSFH, LPVOID, WORD );
EXTERNC WORD   APIENTRY DosWrite     ( HDOSFH, LPVOID, WORD );
EXTERNC HDOSFH APIENTRY DosCloseFile ( HDOSFH );
#endif

```

The one include file contains several sections. The first does a general include of `string.h`, `stdio.h` and `malloc.h`. These includes are assumed to be needed by all modules. Next, the include for this book, `book.h`, is included.

When programming for Windows, make sure that `windows.h` is included before `book.h` gets included and make sure that you are using either `/GA` (for applications) or `/GD` (for DLLs) on the Microsoft C8 command line. This

ensures that `_WINDOWS` gets defined.

The `NEWHANDLE` section comes next. For every new module that you write, you will need to add a `NEWHANDLE(HOBJ)` line. The `OBJ` part is a short descriptive name for the module you are implementing.

The last section of this file is the `USE_*` section. For every `NEWHANDLE(HOBJ)` added to this file, you will also add a `#ifdef USE_HOBJ` section that prototypes the interface to the new module you have written.

## A.2 BOOK.H Include File

`BOOK.H` is the include file for everything that this book describes. You should never include this file directly. Instead, it should be included through `APP.H` as described in the previous section.

`BOOK.H` assumes that you have a standard C compiler. If you do not have a standard C compiler, you may still be OK if your C preprocessor follows the Reiser model ([§2.2.8](#)). The problem is the usage of the stringizing operator (`#`) and the token pasting operator (`##`). See [§2.2.7](#) for more information on preprocessor operators.

```
BOOK.H
/*--- If target is C8 segmented architecture ---*/
#if (defined(_MSC_VER) && defined(MSDOS))
#define FAR _far
#define NEAR _near
#define FASTCALL _fastcall
#define PASCAL _pascal
#define EXPORT _export
#define BASEDIN(seg) _based(_seaname(#seg))

/*--- Else assume target is flat memory model ---*/
#else
#define FAR
#define NEAR
#define FASTCALL
#define PASCAL
#define EXPORT
#define BASEDIN(seg)
#endif
```

```

/*--- size_t mapping ---*/
#define SIZET size_t

/*--- Useful defines ---*/
#ifndef _WINDOWS
typedef char FAR*LPSTR;
typedef void FAR*LPVOID;
typedef unsigned short WORD;
typedef int BOOL;
#define FALSE (0)
#define TRUE (1)
#endif

/*--- Assert during compiling (not run-time) ---*/
#define CompilerAssert(exp) extern char _CompilerAssert[(exp)?
1:-1]

/*--- TYPEDEF depends upon C/C++ ---*/
#ifdef __cplusplus
#define TYPEDEF
#else
#define TYPEDEF typedef
#endif

/*--- EXTERNC depends upon C/C++ ---*/
#ifdef __cplusplus
#define EXTERNC extern "C"
#else
#define EXTERNC
#endif

/*--- APIENTRY for app/dll ---*/
#ifdef _WINDLL
#define APIENTRY EXPORT FAR PASCAL
#else
#define APIENTRY FAR PASCAL
#endif

/*--- LOCAL/LOCALASM defines ---*/
#define LOCAL static NEAR FASTCALL
#define LOCALASM static NEAR PASCAL

/*--- Other useful defines ---*/
#define CSCHAR static char BASEDIN(_CODE)
#define NewScope

/*--- Absolute value macro ---*/
#define ABS(x) (((x)>0)?(x):- (x))

```

```

/*--- Is a number a power of two ---*/
#define ISPOWER2(x) (!((x)&((x)-1)))

/*--- Number of static elements in an array ---*/
#define NUMSTATICELS(pArray) (sizeof(pArray)/sizeof(*pArray))

/*--- Loop Macros ---*/
#define LOOP(nArg) { int _nMax=nArg; int loop; \
    for (loop=0; loop<_nMax; ++loop)
#define LLOOP(lArg) { long _lMax=lArg; long lLoop; \
    for (lLoop=0; lLoop<_lMax; ++lLoop)
#define ENDLOOP }

/*--- WinAssert support ---*/
#define USEWINASSERT CSCHAR szSRCFILE[]=__FILE__; \
    BOOL static NEAR _DoWinAssert( int nLine ) { \
        ReportWinAssert(szSRCFILE, nLine); \
        WinAssert(nLine); \
        return(FALSE); \
    }
#define AssertError _DoWinAssert(__LINE__)
#define WinAssert(exp) if (!(exp)) {AssertError;} else
EXTERNC void APIENTRY ReportWinAssert( LPSTR, int );

/*--- What is a class descriptor---*/
typedef struct {
    LPSTR lpVarName;
} CLASSDESC, FAR*LPCLASSDESC;

/*--- Declare a new handle ---*/
#define NEWHANDLE(Handle) typedef struct tag##Handle *Handle

/*--- Class descriptor name from object name ---*/
#define _CD(hObj) hObj##_ClassDesc

/*--- The class macro ---*/
#define CLASS(hObj,Handle) \
    static CLASSDESC _CD(hObj)={#hObj}; TYPEDEF struct \
    tag##Handle

/*--- Object verification macros---*/
#define VERIFY(hObj) WinAssert(_VERIFY(hObj))
#define VERIFYZ(hObj) if (!(hObj)) {} else VERIFY(hObj)

/*--- Object verification helper macros ---*/
#define _S4 (sizeof(LPCLASSDESC))
#define _S8 (sizeof(LPCLASSDESC)+sizeof(LPVOID))
#define _VERIFY(hObj) \
    ( FmIsPtrOk(hObj) && \
    (((LPVOID)hObj)==*(LPVOID FAR*)((LPSTR)hObj-_S8)) \

```

```

        && ( (&_CD(hObj)) == * (LPCLASSDESC FAR*) ( (LPSTR)hObj - _S4) ) )

/*--- Heap manager prototypes ---*/
EXTERNC LPVOID APIENTRY FmNew      ( SIZE_T, LPCLASSDESC, LPSTR,
int);
EXTERNC LPVOID APIENTRY FmFree     ( LPVOID );
EXTERNC LPVOID APIENTRY FmRealloc  ( LPVOID, SIZE_T, LPSTR, int );
EXTERNC LPVOID APIENTRY FmStrDup   ( LPSTR, LPSTR, int );
EXTERNC void   APIENTRY FmWalkHeap ( void );
EXTERNC BOOL   APIENTRY FmIsPtrOk  ( LPVOID );

/*--- NEWOBJ() and FREE() Interface ---*/
#define _LPV(hObj) * (LPVOID FAR*) &hObj
#define NEWOBJ(hObj) \
    ( _LPV(hObj) = FmNew(sizeof(*hObj), &_CD(hObj), szSRCFILE, __LINE__) )
#define FREE(hObj) ( _LPV(hObj) = FmFree(hObj) )

/*--- String interface macros ---*/
#define NEWSTRING(lpDst, wSize) \
    ( _LPV(lpDst) = FmNew((SIZE_T)(wSize), NULL, szSRCFILE, __LINE__) )
#define MYLSTRDUP(lpDst, lpSrc) \
    ( _LPV(lpDst) = FmStrDup(lpSrc, szSRCFILE, __LINE__) )

/*--- Array interface macros ---*/
#define NEWARRAY(lpArray, wSize) \
    ( _LPV(lpArray) = FmNew((SIZE_T)(sizeof(*lpArray) * (wSize)), \
    NULL, szSRCFILE, __LINE__) )
#define SIZEARRAY(lpArray, wSize) \
    ( _LPV(lpArray) = FmRealloc((lpArray), \
    (SIZE_T)(sizeof(*lpArray) * (wSize)), szSRCFILE, __LINE__) )

```

## A.3 HEAP.C Module

This code implements the heap manager as described in [Chapter 5](#).

### HEAP.C

```

/*pm-----
-
*
*  OUTLINE:
*
*    This module REPLACES the memory management routines of
*    the C run-time library. As such, this new interface
*    should be used exclusively.
*

```

```

* IMPLEMENTATION:
*
* A wrapper is provided around all memory objects that
* allows for run-time type checking, symbolic dumps of
* the heap and validation of heap pointers.
*
* NOTES:
*
* - YOU must code an FmIsPtrOk() that works properly for
* your environment.
*
*-----
*/

#include "app.h"

USEWINASSERT

/*--- Heap objects are aligned on sizeof(int) boundaries ---*/
#define ALIGNMENT (sizeof(int))
#define DOALIGN(num) (((num)+ALIGNMENT-1)&~(ALIGNMENT-1))
CompilerAssert(ISPOWER2(ALIGNMENT));

/*--- Declare what LPPREFIX/LPPOSTFIX are ---*/
typedef struct tagPREFIX FAR*LPPREFIX;
typedef struct tagPOSTFIX FAR*LPPOSTFIX;

/*--- Prefix structure before every heap object---*/
typedef struct tagPREFIX {
    LPPREFIX lpPrev;           /* previous object in heap */
    LPPREFIX lpNext;           /* next object in heap */
    LPPOSTFIX lpPostfix;       /* ptr to postfix object */
    LPSTR lpFilename;          /* filename ptr or NULL */
    long lLineNumber;          /* line number or 0 */
    LPVOID lpMem;              /* FmNew() ptr of object */
    LPCLASSDESC lpClassDesc;   /* class descriptor ptr or NULL */
} PREFIX;

/*--- Postfix structure after every heap object ---*/
typedef struct tagPOSTFIX {
    LPPREFIX lpPrefix;
} POSTFIX;

/*--- Verify alignment of prefix structure ---*/
CompilerAssert(!(sizeof(PREFIX)%ALIGNMENT));

/*--- Points to first object in linked list of heap objects ---*/
static LPPREFIX lpHeapHead=NULL;

/*--- Local prototypes ---*/

```

```

void LOCAL AddToLinkedList      ( LPPREFIX );
void LOCAL RemoveFromLinkedList ( LPPREFIX );
BOOL LOCAL VerifyHeapPointer    ( LPVOID );
void LOCAL RenderDesc           ( LPPREFIX, LPSTR );

/*pf-----
-
*
* DESCRIPTION: (Far Memory New)   JLJ
*
* Allocate a new block of memory from the heap.
*
* ARGUMENTS:
*
*   wSize      - Size of object to allocate
*   lpClassDesc - Class descriptor for object (or NULL)
*   lpFile      - Filename where object was allocated
*   nLine      - Line number where object was allocated
*
* RETURNS:
*
*   A long pointer to the memory object or NULL
*
*-----
*/

LPVOID APIENTRY FmNew( SIZE_T wSize, LPCLASSDESC lpClassDesc,
    LPSTR lpFile, int nLine )
{
    LPPREFIX lpPrefix;
    wSize = DOALIGN(wSize);
    lpPrefix=
(LPPREFIX)malloc(sizeof(PREFIX)+wSize+sizeof(POSTFIX));
    if (lpPrefix) {
        AddToLinkedList( lpPrefix );
        lpPrefix->lpPostfix = (LPPOSTFIX)((LPSTR)(lpPrefix+1)+wSize);
        lpPrefix->lpPostfix->lpPrefix = lpPrefix;
        lpPrefix->lpFilename = lpFile;
        lpPrefix->lLineNumber = nLine;
        lpPrefix->lpMem = lpPrefix+1;
        lpPrefix->lpClassDesc = lpClassDesc;
        memset( lpPrefix->lpMem, 0, wSize );
    }
    else {
        AssertError;                /* Report out of memory error */
    }
    return(lpPrefix ? lpPrefix+1 : NULL);
} /* FmNew */

```



```

/*pf-----
-
*
* DESCRIPTION: (Far Memory Free)   JLJ
*
*   Free a block of memory that was previously allocated
*   through FmNew().
*
* ARGUMENTS:
*
*   lpMem - Heap pointer to free or NULL
*
* RETURNS:
*
*   NULL
*
*-----
*/

```

```

LPVOID APIENTRY FmFree( LPVOID lpMem )
{
    if (VerifyHeapPointer(lpMem)) {
        LPPREFIX lpPrefix=(LPPREFIX)lpMem-1;
        SIZET wSize=(LPSTR)(lpPrefix->lpPostfix+1)-
(LPSTR)lpPrefix;
        RemoveFromLinkedList( lpPrefix );
        memset( lpPrefix, 0, wSize );
        free(lpPrefix);
    }
    return (NULL);
} /* FmFree */

```

```

/*pf-----
-
*
* DESCRIPTION: (Far Memory String Dup)   JLJ
*
*   Helper function for the MYLSTRDUP() macro
*
* ARGUMENTS:
*
*   lpS      - String to duplicate (or NULL)
*   lpFile   - Filename where string is being duplicated
*   nLine    - Line number where string is being duplicated
*
* RETURNS:

```

```

*
*   A pointer to the duplicated string or NULL
*
*-----
*/

LPVOID APIENTRY FmStrDup( LPSTR lpS, LPSTR lpFile, int nLine )
{
    LPVOID lpReturn=NULL;

    if (lpS) {
        SIZET wSize = (SIZET)(strlen(lpS)+1);
        lpReturn = FmNew( wSize, NULL, lpFile, nLine );
        if (lpReturn) {
            memcpy( lpReturn, lpS, wSize );
        }
    }
    return(lpReturn);
} /* FmStrDup */

/*pf-----
-
*
*   DESCRIPTION: (Far Memory Realloc)   JLJ
*
*   Reallocate a block of memory
*
*   ARGUMENTS:
*
*   lpOld   - Heap object to reallocate or NULL
*   wSize   - New size of the object
*   lpFile  - Filename where realloc is taking place
*   nLine   - Line number where realloc is taking place
*
*   RETURNS:
*
*   A pointer to the reallocated memory or NULL
*
*-----
*/

LPVOID APIENTRY FmRealloc( LPVOID lpOld, SIZET wSize,
    LPSTR lpFile, int nLine )
{
    LPVOID lpNew=NULL;

    /*--- Try to realloc ---*/
    if (lpOld) {

```

```

    if (VerifyHeapPointer(lpOld)) {
        LPPREFIX lpPrefix=(LPPREFIX)lpOld-1;
        LPPREFIX lpNewPrefix;
        LPPREFIX lpPre;

        /*--- Try to reallocate block ---*/
        RemoveFromLinkedList( lpPrefix );
        memset( lpPrefix->lpPostfix, 0, sizeof(POSTFIX) );
        wSize = DOALIGN(wSize);
        lpNewPrefix=(LPPREFIX)realloc(lpPrefix,
            sizeof(PREFIX)+wSize+sizeof(POSTFIX));

        /*--- Add new (or failed old) back in ---*/
        lpPre=(lpNewPrefix?lpNewPrefix:lpPrefix);
        AddToLinkedList( lpPre );
        lpPre->lpPostfix = (LPPOSTFIX)((LPSTR)(lpPre+1)+wSize);
        lpPre->lpPostfix->lpPrefix = lpPre;
        lpPre->lpMem = lpPre+1;

        /*--- Finish ---*/
        lpNew = (lpNewPrefix ? &lpNewPrefix[1] : NULL);
        if (!lpNew) {
            /* Report out of memory error */
            AssertError;
        }
    }

    /*--- Else try new allocation ---*/
    else {
        lpNew = FmNew( wSize, NULL, lpFile, nLine );
    }

    /*--- Return address to object ---*/
    return(lpNew);
} /* FmRealloc */

```

```

/*pf-----
-
*
* DESCRIPTION: (Walk Heap)  JLJ
*
* Display a symbolic dump of the heap by walking the
* heap and displaying all objects in the heap.
*
* ARGUMENTS:
*
* (void)

```

```

*
* RETURNS:
*
* (void)
*
*-----
*/

void APIENTRY FmWalkHeap( void )
{
    if (lpHeapHead) {
        LPPREFIX lpCur=lpHeapHead;
        while (VerifyHeapPointer(&lpCur[1])) {
            char buffer[100];
            RenderDesc( lpCur, buffer );
            /*--- print out buffer ---*/
            /* printf( "walk: %s\n", buffer ); */
            lpCur = lpCur->lpNext;
            if (lpCur==lpHeapHead) {
                break;
            }
        }
    }
}

/* FmWalkHeap */

/*p-----
-
*
* DESCRIPTION: (Add Heap Object to Linked List)  JLJ
*
* Add the given heap object into the doubly linked list
* of heap objects.
*
* ARGUMENTS:
*
* lpAdd - Prefix pointer to heap object
*
* RETURNS:
*
* (void)
*
*-----
*/

void LOCAL AddToLinkedList( LPPREFIX lpAdd )
{
    /*--- Add before current head of list ---*/
    if (lpHeapHead) {

```

```

        lpAdd->lpPrev = lpHeapHead->lpPrev;
        (lpAdd->lpPrev)->lpNext = lpAdd;
        lpAdd->lpNext = lpHeapHead;
        (lpAdd->lpNext)->lpPrev = lpAdd;
    }

    /*--- Else first node ---*/
    else {
        lpAdd->lpPrev = lpAdd;
        lpAdd->lpNext = lpAdd;
    }

    /*--- Make new item head of list ---*/
    lpHeapHead = lpAdd;
} /* AddToLinkedList */

/*p-----
-
*
* DESCRIPTION: (Remove Heap Object from Linked List)  JLJ
*
* Remove the given heap object from the doubly linked list
* of heap objects.
*
* ARGUMENTS:
*
* lpRemove - Prefix pointer to heap object
*
* RETURNS:
*
* (void)
*-----
*/

void LOCAL RemoveFromLinkedList( LPPREFIX lpRemove )
{
    /*--- Remove from doubly linked list ---*/
    (lpRemove->lpPrev)->lpNext = lpRemove->lpNext;
    (lpRemove->lpNext)->lpPrev = lpRemove->lpPrev;

    /*--- Possibly correct head pointer ---*/
    if (lpRemove==lpHeapHead) {
        lpHeapHead = ((lpRemove->lpNext==lpRemove) ? NULL :
            lpRemove->lpNext);
    }
}

```

```

} /* RemoveFromLinkedList */

/*p-----
-
*
* DESCRIPTION: (Verify Heap Pointer)  JLJ
*
* Verify that a pointer points into that heap to a valid
* object in the heap.
*
* ARGUMENTS:
*
* lpMem - Heap pointer to validate
*
* RETURNS:
*
* Heap pointer is valid (TRUE) or not (FALSE)
*-----
*/

BOOL LOCAL VerifyHeapPointer( LPVOID lpMem )
{
    BOOL bOk=FALSE;

    if (lpMem) {
        WinAssert(FmIsPtrOk(lpMem)) {
            LPPREFIX lpPrefix=(LPPREFIX)lpMem-1;
            WinAssert(lpPrefix->lpMem==lpMem) {
                WinAssert(lpPrefix->lpPostfix->lpPrefix==lpPrefix) {
                    bOk = TRUE;
                }
            }
        }
    }

    return (bOk);
} /* VerifyHeapPointer */

/*pf-----
-
*
* DESCRIPTION: (Does Pointer Point into the Heap)  JLJ
*
* Does the given memory pointer point anywhere into
* the heap.
*

```

```

* ARGUMENTS:
*
*     lpMem - Heap pointer to check
*
* RETURNS:
*
*     Pointer points into the heap (TRUE) or not (FALSE)
*
*-----
*/

#if (defined(_WINDOWS) || defined(_WINDLL))
BOOL APIENTRY FmIsPtrOk( LPVOID lpMem )
{
    BOOL bOk=FALSE;
    _asm xor ax, ax                ;; assume bad selector
    _asm lsl ax, word ptr [lpMem+2] ;; get selector limit
    _asm cmp word ptr [lpMem], ax   ;; is ptr offset under limit
    _asm jae done                  ;; no, bad pointer
    _asm mov bOk, 1                ;; yes, pointer OK
    _asm done:
    return (bOk);
} /* FmIsPtrOk */
#else
BOOL APIENTRY FmIsPtrOk( LPVOID lpMem )
{
    return ((lpMem) && (!((long)lpMem&(ALIGNMENT-1))));
} /* FmIsPtrOk */
#endif

/*p-----
-
*
* DESCRIPTION: (Render Description of Heap Object)   JLJ
*
*     Render a text description for the given heap object.
*
* ARGUMENTS:
*
*     lpPrefix - Prefix pointer to heap object
*     lpBuffer - Where to place text description
*
* RETURNS:
*
*     (void)
*
*-----

```

```

*/
void LOCAL RenderDesc( LPPREFIX lpPrefix, LPSTR lpBuffer )
{
    if (lpPrefix->lpMem==&lpPrefix[1]) {
        sprintf( lpBuffer, "%08lx ", lpPrefix );
        if (lpPrefix->lpFilename) {
            sprintf( lpBuffer+strlen(lpBuffer), "%12s %4ld ",
                    lpPrefix->lpFilename, lpPrefix->lLineNumber );
        }
        if (lpPrefix->lpClassDesc) {
            sprintf( lpBuffer+strlen(lpBuffer), "%s",
                    lpPrefix->lpClassDesc->lpVarName );
        }
    }
    else {
        strcpy( lpBuffer, "(bad)" );
    }
}

/* RenderDesc */

```

## A.4 RAND.C Module

This code implements the random number generator module as described in [Chapter 4](#).

### Random number generator

```

/*pm-----
-
*
*  OUTLINE:
*
*      This module implements a random number object.
*
*  IMPLEMENTATION:
*
*      The random numbers are implemented just like they are
*      in the Microsoft C8 RTL.
*
*  NOTES:
*
*-----
*/

```



```

#define USE_HRAND

#include "app.h"

USEWINASSERT

/*--- The random number class object ---*/
CLASS(hRand, HRAND) {
    long lRand;
};

/*--- Implement random numbers just like the MS C8 RTL ---*/
#define NEXTRAND(l)  (l*214013L+2531011L)
#define FINALRAND(l) ((l>>16)&0x7FFF)

/*pf-----
-
*
* DESCRIPTION: (Create Random Number Object)  JLJ
*
* Given a seed, create a new random number generator object
*
* ARGUMENTS:
*
* nSeed - Seed of the new random number generator
*
* RETURNS:
*
* A new random number generator object
*
*-----
*/

HRAND APIENTRY RandCreate( int nSeed )
{
    HRAND hRand;
    NEWOBJ(hRand);
    hRand->lRand = nSeed;
    return (hRand);
} /* RandCreate */

/*pf-----
-
*
* DESCRIPTION: (Destroy Random Number Object)  JLJ
*
* Destroy the given random number generator object

```

```

*
* ARGUMENTS:
*
*   hRand - Random number generator object or NULL
*
* RETURNS:
*
*   NULL
*
*-----
*/

HRAND APIENTRY RandDestroy( HRAND hRand )
{
    VERIFYZ(hRand) {
        FREE(hRand);
    }
    return (NULL);
} /* RandDestroy */

/*pf-----
-
*
* DESCRIPTION: (Generate Next Random Number)   JLJ
*
*   Generate the next random number for the given random
*   number generator object.
*
* ARGUMENTS:
*
*   hRand - Random number generator object
*
* RETURNS:
*
*   The next random number
*
*-----
*/

int APIENTRY RandNext( HRAND hRand )
{
    int nRand=0;
    VERIFY(hRand) {
        hRand->lRand = NEXTRAND(hRand->lRand);
        nRand = (int)FINALRAND(hRand->lRand);
    }
    return(nRand);
}

```

```
} /* RandNext */
```

## A.5 DOS.C Module

This code implements the DOS module as described in [Chapter 6](#).

### A DOS interface

```
/******  
/*  
/*          (project name)          */  
/*  
/*  Copyright (date) (Company Name). All rights reserved.  */  
/*  
/*  This program contains the confidential trade secret    */  
/*  information of (Company Name). Use, disclosure, or      */  
/*  copying without written consent is strictly prohibited. */  
/*  
/******  
  
/*pm-----  
*  
*  OUTLINE:  
*  
*    This module provides access to the low-level file I/O  
*    functions of the standard Microsoft C run-time library.  
*  
*  IMPLEMENTATION:  
*  
*    This module is simply a code wrapper module.  
*  
*  NOTES:  
*  
*-----*/  
  
#define USE_LOWIO  
#define USE_HDOSFH  
  
#include "app.h"  
  
USEWINASSERT  
  
/*--- The class object ---*/  
CLASS(hDosFh, HDOSFH) {  
    int fh;
```

```

    };

/*pf-----
*
*   DESCRIPTION: (Open File)   JLJ
*
*   Attempt to open a file
*
*   ARGUMENTS:
*
*   lpFilename - The name of the file to open
*
*   RETURNS:
*
*   A file object handle or NULL if there was some error
*   in opening the specified file.
*
*-----*/

HDOSFH APIENTRY DosOpenFile( LPSTR lpFilename )
{
    HDOSFH hDosFh=NULL;
    int fh=open(lpFilename, _O_RDWR|_O_BINARY);
    if (fh!=-1) {
        NEWOBJ(hDosFh);
        hDosFh->fh = fh;
    }
    return (hDosFh);
} /* DosOpenFile */

/*pf-----
*
*   DESCRIPTION: (Close File)   JLJ
*
*   Close a previously opened file
*
*   ARGUMENTS:
*
*   hDosFh - The file object or NULL
*
*   RETURNS:
*
*   NULL
*
*-----*/

HDOSFH APIENTRY DosCloseFile( HDOSFH hDosFh )

```

```

{

    VERIFY(hDosFh) {
        int nResult=close(hDosFh->fh);
        WinAssert(!nResult);
        FREE(hDosFh);
    }
    return (NULL);
} /* DosCloseFile */


/*pf-----
*
*   DESCRIPTION: (Read File)   JLJ
*
*   Attempt to read a block of information from a file.
*
*   ARGUMENTS:
*
*   hDosFh - The file object
*   lpMem   - Pointer to memory buffer
*   wCount  - Number of bytes to read into the memory buffer
*
*   RETURNS:
*
*   The number of bytes that were actually read
*
*-----*/

WORD APIENTRY DosRead( HDOSFH hDosFh, LPVOID lpMem, WORD wCount )
{
    WORD wNumRead=0;

    VERIFY(hDosFh) {
        wNumRead = (WORD)read(hDosFh->fh, lpMem, wCount);
    }
    return (wNumRead);
} /* DosRead */


/*pf-----
*
*   DESCRIPTION: (Write File)   JLJ
*
*   Attempt to write a block of information to a file.
*
*   ARGUMENTS:
*

```

```

*   hDosFh - The file object
*   lpMem  - Pointer to memory buffer
*   wCount - Number of bytes to write to the file
*
* RETURNS:
*
*   The number of bytes that were actually written
*
*-----*/
WORD APIENTRY DosWrite( HDOSFH hDosFh, LPVOID lpMem, WORD wCount )
{
    WORD wNumWritten=0;

    VERIFY(hDosFh) {
        wNumWritten = (WORD)write(hDosFh->fh, lpMem, wCount);
    }
    return (wNumWritten);
} /* DosWrite */

```

## A.6 OutputDebugString() for MS-DOS Programmers

If you are programming for Windows, you already have access to the OutputDebugString() function for placing messages onto the monochrome screen. This is an option in the DBWIN.EXE program provided with Microsoft C8.

However, if you are programming under MS-DOS, an OutputDebugString() that you can use to place messages onto the monochrome screen is as follows.

### OutputDebugString() for MS-DOS programmers

```

/*pf-----
-
*
* DESCRIPTION: (Output String to Mono Screen)  JLJ
*
*   Scrolls the monochrome screen and places a new
*   string on the 25'th line.
*
* ARGUMENTS:
*
*   lpS - String to place on monochrome screen

```

```

*
* RETURNS:
*
* (void)
*
*-----
*/

void APIENTRY OutputDebugString( LPSTR lpS )
{
    LPSTR lpScreen=(LPSTR)0xB0000000; /* base of mono screen
*/
    int nPos=0;                        /* for walking lpS string
*/
    /*--- Scroll monochrome screen up one line ---*/
    fmemcpy( lpScreen, lpScreen+2*80, 2*80*24 );
    /*--- Place new line down in 25'th line ---*/
    for (int loop=0; loop<80; ++loop) {
        lpScreen[2*(80*24+loop)] = (lpS[nPos]?lpS[nPos++]:' ');
    }

} /* OutputDebugString */

```

Refer to [§7.27](#) for a description of OutputDebugString().

## A.7 ReportWinAssert()

The ReportWinAssert() functions presented here are bare-bones and should be tailored by you. ReportWinAssert() is called whenever there has been an assertion failure in your code. This can happen when there are run-time type checking failures or when the heap manager is reporting a problem.

 A ReportWinAssert() that can be used for Windows is as follows.

### ReportWinAssert() for Windows

```

void APIENTRY ReportWinAssert( LPSTR lpFilename, int nLine )
{
    char buffer[100];
    wsprintf( buffer, "WinAssert: %s %d", lpFilename, nLine );
    MessageBox( NULL, buffer, "WinAssert", MB_OK|MB_SYSTEMMODAL
);

} /* ReportWinAssert */

```

---

A ReportWinAssert() that can be used for C console apps is as follows.

**ReportWinAssert() for C console apps**

```
void APIENTRY ReportWinAssert( LPSTR lpFilename, int nLine )
{
    printf( "WinAssert: %s-%d (Press Enter) ", lpFilename, nLine
);
    while ('\n'!=getchar()) {
        ;
    }
} /* ReportWinAssert */
```

---

Copyright © 1993-1995, 2002-2003 Jerry Jongerius  
This book was previously published by Person Education, Inc.,  
formerly known as Prentice Hall. ISBN: 0-13-183898-9



# Writing Bug-Free C Code *A Programming*

*Style That Automatically Detects Bugs in C Code*

by Jerry Jongerius / January 1995



## [0. Preface](#)

## [1. Understand Why Bugs Exist](#)

## [2. Know Your Environment](#)

## [3. Rock-Solid Base](#)

## [4. The Class Methodology](#)

## [5. A New Heap Manager](#)

## [6. Designing Modules](#)

## [7. General Tips](#)

## [8. Style Guide](#)

## [9. Conclusion](#)

## [A. Appendix: Code Listings](#)

## [References](#)

## C Programming *with*:

- Class Methodology [§4](#)
- Data hiding [§4.2](#)
- Runtime type checking [§4.4](#)
- Compile time type checking [§4.3](#)
- Fault tolerant asserts [§3.3](#)
- Fault tolerant functions [§4.6](#)
- Compile-time asserts [§2.1.4](#)
- Symbolic heap walking [§5.2.9](#)
- Heap leak detection [§5.5](#)

Book C Source Code (9k): [source.zip](#) (for DOS, Windows, UNIX, etc)



Sections of this book that talk about Microsoft or Windows are generally marked with the 'Windows' graphic. You can safely skip those sections of this book if you want to.

Copyright © 1993-1995, 2002-2003 Jerry Jongerius, [jerryj@duckware.com](mailto:jerryj@duckware.com)

This book was previously published by Person Education, Inc.,  
formerly known as Prentice Hall. ISBN: 0-13-183898-9

[Check Amazon for paperback availability.](#)

# Chapter 2: Know Your Environment [Writing Bug-Free C Code](#)

[2.1 The C Language](#)

[2.2 C Preprocessor](#)

[2.3 Programming Puzzles](#)

[2.4 Microsoft Windows](#)

[2.5 Chapter Summary](#)


Before you can efficiently institute new programming methodologies that help eliminate bugs, you need to understand the features available to you in your programming environment.

Quite often, it is beneficial to ask yourself why a particular feature is present in the environment. If you are already using the feature, great, but is there another way you could also be using it? If you are not using the feature, try to think of why the feature was added, because someone needed and requested the feature. Why did they need the feature and how are they using it?

Try to become an expert in your environment.

In the process of learning about all the features of your environment, you may eventually become an expert on it. And the learning process never stops. With each new version of the tools in your environment, look in the manuals to find out what new features have been added. Sometimes features are even removed!

## 2.1 The C Language

 Microsoft C 8.0 (a.k.a. Microsoft Visual C++ 1.0) comes in two editions: the standard edition and the professional edition. The standard edition is targeted to the part-time programmer, the professional edition to the full-time programmer. The professional edition comes with more manuals and, more important, the Windows 3.1 SDK.

No matter which C compiler you are using, it is important that you fully

understand the C programming environment.

### 2.1.1 The Array Operator

You've used the array operator to index an array before, but have you given any thought to how the compiler interprets the array operator? You may not know how it does, especially if you learned C after learning another language first!

Suppose you have a character array called `buffer` and an integer `nIndex` used to index into the character array. How do you obtain a character from the character array? Most everyone will tell you `buffer[nIndex]`, namely, the array name followed by the array index in brackets.

Did you know that `nIndex[buffer]` also works! This syntax is not recommended, but it does work. Do you know why?

The reason that `buffer[nIndex]` references the same character as `nIndex[buffer]` is clearly stated in §A7.3.1 of [The C Programming Language \(2nd ed.\)](#) as follows:

*A postfix expression followed by an expression in square brackets is a postfix expression denoting a subscripted array reference. One of the two expressions must have type "pointer to T", where T is some type, and the other must have integral type; the type of the subscript expression is T. The expression  $E1[E2]$  is identical (by definition) to  $*((E1)+(E2))$ .*

The key to understanding array references is to understand "The expression  $E1[E2]$  is identical (by definition) to  $*((E1)+(E2))$ ." In terms of the example, `buffer[nIndex]` is identical to  $*((\text{buffer})+(\text{nIndex}))$ . Addition is commutative (i.e.,  $3 + 4$  equals  $4 + 3$ ), which makes  $*((\text{buffer})+(\text{nIndex}))$  identical to  $*((\text{nIndex})+(\text{buffer}))$ , which is `nIndex[buffer]`.

Array reference $E1[E2]$ is identical to $*((E1)+(E2))$ .
---

### 2.1.2 The Structure Pointer Operator

You all know that if `p` is a pointer to a structure, that `p->structure-member` refers to a particular member of the structure. Suppose that you could no longer use the structure pointer operator `->`. Could you somehow use the other operators in C and continue coding?

The answer is yes. Instead of `p->structure-member`, use `(*p).structure-member`.

If `p` is a pointer to a structure, the structure reference `p->member` is identical to `(*p).member`.

Again, this is not a recommended syntax, but it is good to understand that it does work and why it works. It is spelled out in §A7.3.3 of [The C Programming Language](#).



### 2.1.3 Use the Highest Compiler Warning Level

The warning and error messages of the Microsoft C compiler are getting better with each release of the compiler. I strongly recommend that you compile your code at the highest warning level available to you. For Microsoft C8, this is warning level four. The command line option for this is `/W4`.

At warning level four, there may be some warning messages that you want to totally ignore. An example in Microsoft C8 are warnings produced by unused declared inline functions. I declare my inline functions in an include file. The source files then use only those inline functions that are needed. Unfortunately, the unreferenced inline functions produce the unreferenced local function has been removed warning message, which is warning number C4505. To disable this warning, you can use the warning `#pragma` in your include files.

```
Pragma to disable warning number 4505 in Microsoft C8
#pragma warning(disable:4505)
```

Some of the useful, interesting warning messages that you can get from the Microsoft C8 compiler are as show in Table 2-1.

Error Number	Error Message
C4019	empty statement at global scope
C4100	unreferenced formal parameter
C4101	unreferenced local variable
C4701	local variable "identifier" may be used without having been initialized. This warning is given only when compiling with the C8 /Oe global register allocation command-line option.
C4702	unreachable code. This warning is given only when compiling with one of the C8 global optimization options (/Oe, /Og or /Ol)
C4705	statement has no effect
C4706	assignment within conditional expression
C4723	potential divide by 0

Table 2-1: Some interesting Microsoft C8 warning messages.

#### 2.1.4 CompilerAssert()

CompilerAssert() is designed to provide error checking at compile-time for assumptions made by the programmer at design-time. These assumptions must be documented within comments in the code as well, but why not also have a compile-time check made whenever possible?

The reasoning behind this is simple. What if a new programmer is working on a section of code in which an assumption is made and the programmer inadvertently changes the code so that the assumption is now invalid? The bug may not show up for a long time. However, if the problem could have been flagged at compile-time, it would have saved a lot of time and effort.

Use CompilerAssert() to verify design-time assumptions at compile-time.

The trick in designing CompilerAssert() is to do it in such a way so that the compiler catches the error at compile-time and yet does not produce any run-time code.

```
CompilerAssert() define  
#define CompilerAssert(exp) extern char _CompilerAssert[(exp)?  
1:-1]
```

*UPDATE: (exp)?1:-1 used to be (exp)?1:0 but it appears that there are some UNIX C compilers that do not complain about an array of zero length even though the standard says the array size must be greater than zero. So 0 was changed to -1 because every compiler must complain about an array with negative size.*

This CompilerAssert() works because it is really just an array declaration and the array bound for an array declaration must be a constant expression that is greater than zero. This is documented in §A8.6.2 of [The C Programming Language](#). Using extern makes sure that no code is generated.

The argument that you pass to CompilerAssert() should be a boolean expression. This means that it evaluates to zero or one. If it is one, the CompilerAssert() is valid and so is the array declaration. If it is zero, the CompilerAssert() is invalid and so is the array declaration. This will cause the compiler to notify you of an error and the compilation will stop.

An added bonus is that the argument to CompilerAssert() must be able to be evaluated at compile-time because the array bound of the array declaration must be a constant expression. This makes it impossible for you to misuse CompilerAssert() and have it accidentally generate some code.

Another bonus is that CompilerAssert() is not limited to use only in functions or source files. It can even be used in include files as needed!

It can also be used more than once in the same scope with no problems. This is due to the usage of extern. Using extern declares the type of

`_CompilerAssert`; it does not define it. Declaring the type of a variable name more than once is OK (as long as the data type is the same). Also, the array size of `(exp)?1:-1` changes any non-zero `(exp)` to one. If this were not done and two different `(exp)` values were used by two `CompilerAssert()`'s, the compiler would complain.

Some linkers may require that a `_CompilerAssert` variable exist while other linkers will optimize out the declared but unused variable. If the linker that you use requires a `_CompilerAssert` variable, place `char _CompilerAssert[1];` in any one of your source files to fix the linker error. The linker that comes with Microsoft C8 does not require this fix.

Let's say that you have coded a function that for some reason requires an internal buffer to be a power of two. Assuming that an [`ISPOWER2\(\)` macro](#) already exists, how would you `CompilerAssert()` this?

```
Sample CompilerAssert() usage
...
char buffer[BUFFER_SIZE];
...
CompilerAssert (ISPOWER2 (sizeof (buffer)) );
...
```

You should test `CompilerAssert()` to verify that it is working in your environment. Under Microsoft C8 and UNIX, the following program will produce a 'negative subscript' error message when compiled.

```
Testing CompilerAssert(), file test.c
#define CompilerAssert(exp) extern char _CompilerAssert[(exp)?1:-1]

int main(void)
{
    /*--- A forced CompilerAssert ---*/
    long lVal;
    CompilerAssert (sizeof (lVal)==0);
    return 0;
} /* main */
```

Compiling test.c under Microsoft C8, should produce an error C2118

```
cl test.c
```

```
test.c(7) : error C2118: negative subscript
```

Compiling test.c under UNIX cc, should produce an error message

```
cc test.c
```

```
test.c: In function `main':
```

```
test.c:7: size of array `_CompilerAssert' is negative
```

## 2.1.5 Variable Declarations

Have you ever wondered why you have to declare `c` is a pointer to a char as `char*c`? Do you know if there is a difference between `char buffer[80]` and `char (*buffer)[80]`? At first, C's declaration syntax may appear difficult, but there is a structure behind it that I am going to try to explain. I firmly believe that the better you understand it, the better C programmer you will be.

What is the result of  $2 + 3 * 5$ ? It is 17, but why? Why did you multiply before you added? The answer is that there is a precedence relationship among the operators. What is the result of  $(2 + 3) * 5$ ? It is 25 because the parentheses override the natural precedence relationship. There is a direct analogy to reading C variable declarations.

You have basically four things to look for: First, parentheses, (...); second, arrays, [...]; third, functions, (); fourth, pointers, \*. The parentheses, just as in expressions, can override precedence relationships. Arrays [] and functions () have a higher precedence than pointers \*. Since arrays and functions always appear to the right of a variable name in a declaration and pointers always appear to the left of the variable name, this in effect tells you that you always move right before moving left when reading the variable declaration.

---



Always try to move right before moving left when reading a variable declaration.

The step-by-step rules for reading most valid variable declarations are as follows:

- Find the variable name in the declaration. Say "variable name is a" and try to move to the right.
- To try to move to the right:
  - a. If in an empty set of parentheses, discard the parentheses and try to move to the right.
  - b. If you find a right parentheses, try to move to the left.
  - c. If you find [count], say "array of count" and try to move to the right.
  - d. If you find (args), say "function taking args and returning a" and try to move to the right.
  - e. If you find a semicolon, try to move to the left.
- To try to move to the left:
  - a. If you find an asterisk, say "pointer to a" and try to move to the right.
  - b. If you find a data type, say "data type" and try to move to the right.

**Sample variable declaration one**

```
char (*buffer)[80];
```

```
  ^   ^   ^       ^  
  4   2   1       3
```

1. buffer is a
2. pointer to an
3. array of 80
4. char

In English, the translation goes like this. You start with char (\*buffer)[80]

with the caret ^ indicating your position. Find the variable name buffer and say (1) buffer is a. You now are left with char (\*^)[80]. You move to the right and find a right parenthesis, so you move to the left, find an asterisk and say (2) pointer to an. You are now left with char (^)[80]. You move to the right and find that you are in an empty set of parentheses so you now discard them. You are now left with char ^[80]. You move to the right and find an array, so you say (3) array of 80 and try to move to the left. You are now left with char, a data type, so you say (4) char.

**Sample variable declaration two**

```
int  (*(*testing)(int))[10];
^    ^ ^    ^      ^    ^
6    4 2    1      3    5
```

1. testing is a
2. pointer to a
3. function taking an integer and returning a
4. pointer to an
5. array of ten
6. integers

**Sample variable declaration three: signal function**

```
void (*signal(int, void (*)(int)))(int);
^    ^ ^    ^      ^    ^    ^    ^
8    6 1    2      5    3    4      7
```

1. signal is a
2. function taking an integer for the first argument and a
3. pointer to a
4. function taking an integer and returning a
5. void for the second argument. This signal function returns a
6. pointer to a
7. function taking an integer and returning a
8. void

Going back to the question at the beginning of this section, do you know if there is a difference between `char buffer[80]` and `char (*buffer)[80]`? What is `buffer`? Does it look as if `buffer` is a pointer to an array of 80 characters in both cases?

In the first case, `buffer` is an array of 80 characters. In the second case, `buffer` is a pointer to an array of 80 characters. It may help clarify the problem if you ask "What is the data type of `*buffer`?" in both cases. In the first case, it is a character. In the second case, it is an array of 80 characters. So there is a seemingly slight but quite significant difference.

This is just an introduction to variable declarations. I hope that you have a new insight into how to read declarations and urge you to look into the topic further.

### 2.1.6 Typedef's Made Easy

I can remember when I first started to learn C that I had difficulty with typedef's. I do not know exactly why, but one day it just hit me. Take any valid variable definition, add typedef to the front of it and the variable name is now the data type.

```
PSTR is a variable that is a character pointer  
char *PSTR;
```

```
PSTR is a new type that is a character pointer  
typedef char *PSTR;
```

### 2.1.7 Name Space

It is possible in C and C++ to spell a typedef name and a variable name

exactly the same. Consider the following.

```
Code that works, but is a bad programming practice
typedef int x;
int main(void)
{
    x x;
    return 0;
} /* main */
```

While some people may consider this neat, it leads to programs that can be hard to read and understand. My advice to you is to assume that everything has a unique name.

Keep all the names in your programs unique.

If you follow the programming methodologies described in later chapters, you will never have the possibility of name collisions.

### 2.1.8 Code Segment Variables

An interesting feature of the Microsoft C8 compiler is the ability to place read-only variables in the code segment. Read/Write variables are not permitted since writing to a code segment is an access violation, resulting in a general protection fault. A set of macros that help to place read-only strings in the code segment are as follows.

```
CSCHAR define
#define BASEDIN(seg) _based(_seaname(#seg))
#define CSCHAR static char BASEDIN(_CODE)
```

CSCHAR (code segment char) may be used just as a char would be used. Notice the usage of static in the #define. This indicates that the variable is allocated permanent storage and that it is known only in the scope in which it is defined.

CSCHAR uses based pointers to place a string in the code segment. Based pointers are not discussed here since there is an excellent discussion of based pointers in the Microsoft C documentation and in the September 1990 [Microsoft Systems Journal article](#) by Richard Shaw.

CSCHAR is a great way of placing read-only strings in the code segment.

For non-segmented architectures, #define the BASEDIN() macro to be nothing. While this does not give you a code segment variable, it does allow you to port the code easily.

```
Using CSCHAR
CSCHAR szFile[]=__FILE__;
...
OutputName(szFile);
```

Another use of code segment variables is for placing read-only data tables in the segment in which they are used. One ideal application of this is in the generation of 16-bit and 32-bit cyclic redundancy checks (CRCs). There are well known table-driven methods for speeding up CRC calculations. If these tables were contained in a data segment, they would use valuable space and would always be in memory. By embedding the tables in the code segment that does the CRC calculations, you free up data segment storage. The code segment is more than likely marked as discardable under Windows; therefore you now have code and read-only data that is read in from the disk and discarded as a set.

### 2.1.9 Adjacent String Literals

Consider the following sample code and the output that is produced.

```
Sample code
#include <stdio.h>
int main(void)
```

```
{
    printf( "String: %s" "One", "Two" "Three" );
    return 0;

} /* main */
```

#### **Output from the sample code**

String: TwoThreeOne

Notice how adjacent string literals are being concatenated in this sample code. The first concatenation is between "String: %s" and "One" to yield "String: %sOne". The second concatenation is between "Two" and "Three" to yield "TwoThree". So the statement really is `printf( "String: %sOne", "TwoThree" );`, which explains the output produced by this code.

In Microsoft C8, string concatenation is done by the compiler, not by the preprocessor. This feature is especially useful for splitting long strings across multiple source lines as in the following example.

#### **Splitting up long strings**

```
#include <stdio.h>
int main(void)
{
    printf( "This is an example of using C's ability to\n"
           "contatenate adjacent string literals. It\n"
           "is a great way to get ONE printf to display\n"
           "a help message.\n" );
    return 0;

} /* main */
```

It is also a good way to place macro arguments in a string.

#### **Placing macro arguments in a string literal**

```
#define PRINTD(var) printf( "Variable " #var "=%d\n", var )
```

PRINTD() works by using the [stringizing operator \(#\)](#) on the macro

argument, namely `#var`. This places the macro argument in quotes and allows the compiler to concatenate the adjacent string literals.

#### **PRINTD() example**

```
#include <stdio.h>
#define PRINTD(var) printf( "Variable " #var "=%d\n", var )
int main(void)
{
    int nTest=10;
    PRINTD(nTest);
    return 0;
} /* main */
```

#### **PRINTD() example output**

```
Variable nTest=10
```



### **2.1.10 Variable Number of Arguments**

The following discussion is specific to Microsoft C8 and the Intel machine architecture. Other compilers and machine architectures may implement function calls and argument passing differently.

C is one of the few languages where passing a variable number of arguments to a function is incredibly easy and incredibly powerful. This mechanism is what is used by `printf()`.

#### **Printf() prototype for Microsoft C8 in stdio.h**

```
int __cdecl printf(const char *, ...);
```

The `...` declaration indicates to the compiler that the following types and number of arguments may vary. It can appear only at the end of an argument list.

Pretend for the moment that you are the compiler writer. How would you implement a variable number of arguments being passed to a function?

Traditional architecture. Consider for a moment what happens at the machine level when a function call is made. A caller pushes onto the stack the arguments that are required by a function and then the function call is made, which pushes the return address onto the stack. You are now executing the function. This function knows that there is a return address on the stack and the number, type and relative stack position of each argument. When the function is ready to return to the caller, it obtains the return address from the stack, removes the arguments from the stack and returns to the caller.

Let us now consider `printf()`. With a variable number of arguments, how does the `printf()` function know how many arguments to pop off the stack when returning from the function? It cannot. The solution is to let the caller restore any arguments pushed onto the stack.

Consider the order in which arguments should be pushed onto the stack. Should the first argument be pushed first or should the last argument be pushed first?

**Printf example**

```
printf( "Testing %s %d", pString, nNum );
```

If the first argument is pushed first (arguments are pushed left to right), the variable argument section is pushed last. In other words, the address of the format string is pushed, then a variable number of arguments is pushed and finally the function call is made, which pushes the return address. The problem is that the relative stack location of the format string address changes depending upon the number of arguments. (See Figure 2-1).



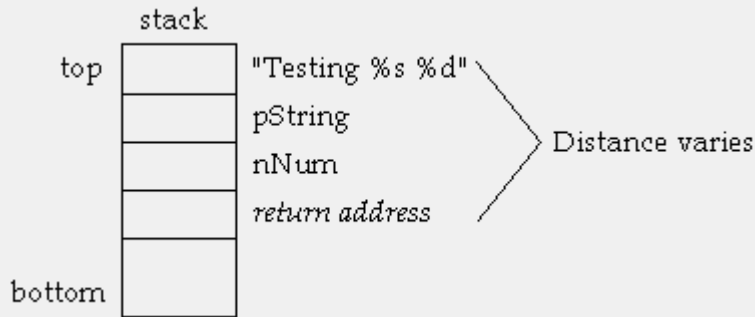


Figure 2-1: Pushing function arguments left to right.

If the last argument is pushed first (arguments are pushed right to left), the variable argument section is pushed first. In other words, the variable number of arguments are pushed right to left, then the address of the format string is pushed and finally the function call is made, which pushes the return address. The relative stack location of the format string address is now adjacent to the return address. (See Figure 2-2).

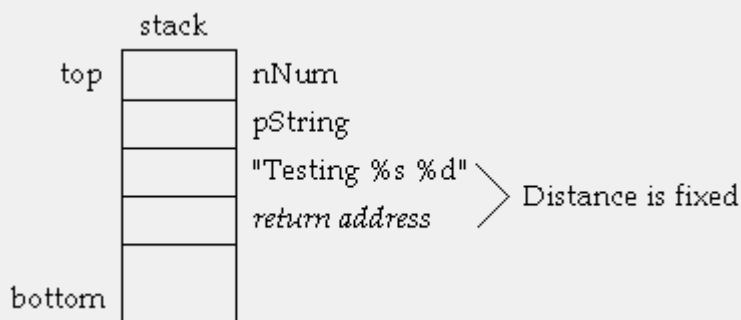



Figure 2-2: Pushing function arguments right to left.

The solution. In order to support passing a variable number of arguments to functions, it appears that the arguments to a function must be pushed right to left and that the caller, not the callee, is responsible for removing the pushed arguments from the stack.

 **2.1.11 Calling Conventions** Were you aware there is more than one way for a function call in Microsoft C8 to be implemented? Each method has advantages and disadvantages.

*C calling convention.* When a function call is made using the C calling convention, the arguments are pushed onto the stack right to left and the caller is responsible for removing the pushed arguments. This convention

was designed to allow for a variable number of arguments to be passed into a function, which can be an incredibly valuable tool (e.g., the `printf()` function). The disadvantage is that the instructions needed to clean up the stack are performed after every call to the function. If you are making a lot of calls in your program, this extra code space adds up.

*Pascal calling convention.* When a function call is made using the Pascal calling convention, the arguments are pushed onto the stack left to right and the callee is responsible for removing the pushed arguments. This convention does not allow for a variable number of arguments. It does, however, conserve code space since the stack is cleaned up by the callee and not the caller. Therefore the cleanup is performed only once. This calling convention is used by all the Windows API functions, except for `DebugOutput()` and `wsprintf()`, which follow the C calling convention.

*Fastcall calling convention.* When a function call is made using the fastcall calling convention, an attempt is made by the compiler to pass as many arguments as possible through the CPU's registers. Those arguments that cannot be passed through registers are passed to the function following the Pascal calling convention. There are restrictions on when this calling convention can be used. Refer to the Microsoft C8 reference manual for more information. This calling convention is used for all functions that are local (private) to a module (see [§6.6.7](#)).



### **2.1.12 Code Generation**

It is sometimes incredibly valuable to see the code that the Microsoft C8 compiler is producing. To do this, use the `/Fc` command line option. The resulting `.cod` file contains the mixed source code and assembly code produced.

The primary reason to do this is to make sure that the programming methodologies that you institute are not adding an abnormal amount of processing-time overhead to the code.

In a few rare circumstances, you can use the code generation option to track down compiler bugs.

I use this option at times because I am just curious to see how good the optimizing compiler is. Sometimes I could swear at the compiler and other times I am amazed at what it is able to do. Microsoft C8 can really do a great job of code optimization. An example that I like to show is the absolute value macro.

#### **Absolute value macro**

```
#define ABS(x) ((x)>0)?(x):- (x) )
```

When the ABS() macro is used on a 16-bit integer, the generated code without optimizations looks like the following.

#### **Code generated by absolute value macro without optimizations**

```
    cmp variable reference,OFFSET 0    ;; is number negative?
    jle L1                             ;; .yes, handle neg
number
    mov ax, variable reference         ;; no, get number
    jmp L2                             ;; .and exit
L1:  mov ax, variable reference         ;; get negative number
    neg ax                             ;; .and make positive
L2:
```

When the ABS() macro is used on a 16-bit integer, the generated code with optimizations looks like the following.

#### **Code generated by absolute value macro with /Os optimizations**

```
1.  mov  ax, variable reference        ;; get number
2.  cwd                                ;; sign extend
3.  xor  ax,dx                         ;; if number is positive,
do
4.  sub  ax,dx                         ;; .nothing, otherwise
negate
```

This optimized assembly code is a work of art. In step 1, the variable is moved into register ax. Step 2 sign extends ax into dx. So, if the number is

positive, dx gets set to 0; otherwise the number is negative and dx gets set to all bits turned on (0xFFFF; negative 1). Provided the number is positive, dx contains 0, so steps 3 and 4 leave the number unchanged. However, if the number is negative, dx contains 0xFFFF, so steps 3 and 4 perform a two's complement, which negates the number. Not bad!

### 2.1.13 Dangling else Problem

Consider the following code.

**Code sample showing dangling else problem**

```
if (test1)
    if (test2) {
        ...
    }
else {
    ...
}
```

The problem with this code is that the else does not belong to the test1 if statement, but instead it belongs to the test2 if statement. An even bigger problem is that the code may have worked at some point in the past before a maintenance programmer included the test2 if statement which previously did not exist. The solution to the immediate problem is simple, however; add a begin and end brace to the test1 if statement as follows.

**Code sample rewritten to eliminate dangling else problem**

```
if (test1) {
    if (test2) {
        ...
    }
}
else {
    ...
}
```

A programming methodology that I have instituted to completely eliminate

any dangling else problems is to say that every if and every else must have begin and end braces. No exceptions.

Every if block of code and else block of code must have begin and end braces.

### 2.1.14 Problems with strncpy()

A function in the C library that I never liked much is strncpy(). Do you know what this function does? Do you know what is going to happen in the following program?

```
strncpy() problem, but code may still work
#include <stdio.h>
#include <string.h>

int main(void)
{
    char buffer[11];
    strncpy( buffer, "hello there", sizeof(buffer) );
    printf( "%s", buffer );
    return 0;
} /* main */
```


strncpy(string1, string2, n) works by copying n characters of string2 to string1. However, if there is no room in string1 for the null character at the end of string2, it is not copied into the string1 buffer. In other words, the buffer is not properly null terminated. If n is greater than the length of string2, string1 is padded with null characters up to length n.

There are two major problems with strncpy. The first is that string1 may not be properly null terminated. The second is the execution overhead of null padding.

In the above example, there is no terminating null for the string copied into buffer. Who knows what the printf() in the example prints out?

The problem is even worse. Depending upon your computer architecture, the above example may still work every time! It all depends upon the alignment of data types. In most cases, 12 bytes are allocated for buffer instead of 11 due to data alignment concerns of the underlying CPU. If this extra byte just happens to be the null character, the code works. If not, the code fails by printing out more than it should in the `printf()` statement. It will just keep on printing characters until it reaches a null character.

Replace questionable and confusing library functions. `Strncpy()` is an example of a bad function.

 A `strncpy()` bug existed in the STARTDOC printer escape under Windows 2.x (STARTDOC tells the system that you are starting to print a new document and it also names the document). If you passed in a string larger than the internal buffer, which was 32 bytes, the buffer did not get properly null terminated.

### 2.1.15 Spell Default Correctly

When you code a switch statement, be careful not to misspell default, because if you do, the compiler will not complain! Your misspelled version of default is considered by the compiler to be a label.

A misspelled default in a switch is considered a label.

Remember that a label is any text you choose followed by a colon. There is no way for the compiler to know that you misspelled default.

### 2.1.16 Logical Operators Use Short-Circuit Logic

Expressions that are connected by the logical and (`&&`) and logical or (`||`) are evaluated left to right and the evaluation of the expression stops once the result can be fully determined.

For example, given `((A) && (B))`, expression A is evaluated first and if zero (false), the result of the entire expression is known to be false so B is

not evaluated. B is evaluated if and only if A is non-zero (true).

Given  $((A) \parallel (B))$ , expression A is evaluated first and if non-zero (true), the result of the entire expression is known to be true so B is not evaluated. B is evaluated if and only if A is zero (false).

Exiting out of a logical expression early because the final expression value is known is called short-circuit logic. Consider the following example.

#### Short-circuit logic example

```
if ((lpMem) && (*lpMem=='A')) {  
    (code body)  
}
```

In this example, lpMem is checked first. If and only if lpMem is non-null does the second check ( $*lpMem == 'A'$ ) take place. This is important because if both checks took place and lpMem was null, the  $*lpMem$  indirection would more than likely cause a fault and the operating system would halt the program.

### 2.1.17 De Morgan's Laws

Do you know how to simplify  $!(a \parallel b)$ ? The answer is  $(a \&\& b)$ . Being able to simplify a complicated expression can be important when trying to understand code that someone else has written. Sometimes being able to not a complicated expression to understand what the expression is not helps you understand the expression.

#### De Morgans laws

1.  $!(a \&\& b) == (!a \parallel !b)$
2.  $!(a \parallel b) == (!a \&\& !b)$

One simple rule applies to De Morgans Laws. To not a logical expression, not both terms and change the logical operator. This helps greatly when you have a complicated logical expression where one of the terms is another

logical expression. To not a term that is itself a logical expression, simply reapply the rule.

To not a logical expression, not both terms and change the logical operator. Apply this rule recursively as needed.

## 2.2 C Preprocessor

### 2.2.1 Writing Macros

How do you write macros? Are you aware of the problems that macros can have? Let's take the following SQUARE() macro.

```
SQUARE() macro, has problems  
#define SQUARE(x) x*x
```

What is SQUARE(7)? It is 49. What is SQUARE(2+3)? Is it 25? No, it is 11. Expanding the macro gives 2+3\*2+3. The multiplication is done first, so now you can see why the result is 11. You may now be inclined to rewrite the SQUARE() macro as follows.

```
SQUARE() macro, has problems  
#define SQUARE(x) (x) * (x)
```

This macro may still have problems with the unary operators. Consider sizeof SQUARE(10), which is really sizeof (10)\*(10), which is not sizeof((10)\*(10))! The correct way to write SQUARE() is as follows.

```
SQUARE() macro, final form  
#define SQUARE(x) ((x) * (x))
```

Notice the extra set of parentheses. This too has problems in some special



circumstances. Namely, how does `SQUARE(++x)` behave? It is actually undefined because it is up to the compiler vendor to decide exactly when multiple post- / pre- increment/decrement operations take place in relation to each other in the entire statement. Let's say `x` contains three; what is `((++x)*(++x))`? Is it 16, 20 or 25? Or what about `SQUARE(x++)`, which is `((x++)*(x++))`. Is it 9? Is it 12? The answer is compiler specific.

Macros that reference an argument more than once have problems with arguments that have side effects.

A possible solution to the problem of side effects is to use inline functions, a feature of C++. This is OK, but in doing so, you lose the polymorphic behavior of working on multiple data types automatically. This is because inline functions are true functions and you must declare the data types of the arguments. So, you must write a new inline function for every data type that you want the inline function to work with. This admittedly is a little bit of a pain, but it may be worth it in some cases.

The best solution, available only in some C++ environments, is the use of templates. Templates are used to describe how to do something, while not specifying the data type to use. Templates are still new and not included in all C++ development environments. They are not included in Microsoft C8.

### 2.2.2 Using Macros That Contain Scope

Have you ever written a macro that needed its own scope? If so, how did you go about writing it? Consider the following example.

**A macro that needs a scope, has problems**  
`#define POW3(x,y) int i; for(i=0; i<y; ++i) {x*=3;}`

The problem with this macro is that because it uses temporary variable `i`, the body of the macro must be contained within its own scope in C and should be contained in its own scope in C++. Consider the following.

**A macro with scope, has problems**

```
#define POW3(x,y) {int i; for(i=0; i<y; ++i) {x*=3;}}
```

Even this latest fix may have problems as this next example shows.

**Using POW3(), has problems**

```
if (expression)
    POW3(lNum, nPow);
else
    (statement)
```

The problem is the semicolon after POW3. The POW3 macro without the semicolon is a valid statement by itself due to the begin/end braces creating a scope. Adding the semicolon only creates a new statement, which is the problem since the else can then no longer be paired with the if.

So how can a macro, no matter how many statements it contains, be enclosed in its own scope, be treated like one statement and require that it be terminated by a semicolon? The solution to this problem is subtle but very elegant.

**A macro with scope, final form**

```
#define POW3(x,y) do {int i; for(i=0; i<y; ++i) {x*=3;}}
while(0)
```

By using a do/while loop that never loops, the macro body gets its own scope and requires a semicolon after it. Most optimizing compilers will optimize away the loop that never loops.

Using a do/while loop that never loops is a great way to hide the body of a macro within its own scope.

However, some C compilers (Microsoft C8 included) may complain about the constant zero in while(0) when the highest warning level of the compiler is used. If this occurs, you can use a #pragma to disable the

warning.

```
Pragma to disable warning number 4127 in Microsoft C8  
#pragma warning(disable:4127)
```

### 2.2.3 If Statements in Macros

If you write a macro that contains if statements, you must be careful not to have the [dangling else §2.1.13](#) problem previously discussed.

```
Macro containing if/else, has problems  
#define ODS(s) if (bDebugging) OutputDebugString(#s)
```

One solution would be to enclose the body of the macro in a do/while loop that never loops, which would have the added benefit of requiring the macro to be terminated with a semicolon. This version of ODS() has the dangling else problem. However, through careful usage of the if/else statement, you can eliminate the problem.

```
Macro containing if/else, problem solved  
#define ODS(s) if (!bDebugging) {} else OutputDebugString(#s)
```

This version of ODS() no longer has the dangling else problem. The solution is to always rework the macro so that it contains both an if clause and an else clause. Again, a semicolon is required after a macro that uses this new form.

Be careful when writing macros that contain if/else statements not to end the macro with an ending brace, or the macro terminated by a semicolon will actually be treated like two statements.

Never conclude a macro with an ending brace.
--

## 2.2.4 Ending a Macro with a Semicolon or a Block of Code

It is possible to use the subtleties of an if/else statement in a macro to your advantage. For example, how would you write a macro that must either be terminated with a semicolon or a block of code?

The [WinAssert\(\)\\_§3.3](#) macro uses this technique to allow a block of code to be conditionally executed.

### **WinAssert() syntax**

```
/*--- Ended with a semicolon ---*/  
WinAssert(expression);  
/*--- Or ended with a block of code ---*/  
WinAssert(expression) {  
    (block of code)  
}
```

Notice that the WinAssert() syntax allows either a semicolon or a block of code to follow it. The WinAssert() macro looks like the following.

### **WinAssert() macro**

```
#define WinAssert(exp) if (!(exp)) {AssertError;} else
```

The key to this WinAssert() macro is that whatever follows the macro is what follows the else statement. A semicolon or a block of code are both valid in this context.

## 2.2.5 LOOP(), LLOOP() and ENDLOOP Macros

The vast majority of the loops that I have written start at zero, have a less than comparison with some upper limit and increment the looping variable by one every iteration. Since this is the case, why not abstract this out of the code into a macro? In addition, since the looping variable is used to control the loop, it should not be visible (accessible) outside the loop. The solution involves three macros. The first two, LOOP() and LLOOP(), set up the for loop for int's and long's. The third macro, ENDLOOP, finishes what

LOOP() and LLOOP() started.

**The LOOP(), LLOOP() and ENDLOOP macros**

```
#define LOOP(nArg) { int _nMax=nArg; int loop; \  
    for (loop=0; loop<_nMax; ++loop)  
#define LLOOP(lArg) { long _lMax=lArg; long lLoop; \  
    for (lLoop=0; lLoop<_lMax; ++lLoop)  
#define ENDLOOP }
```

Notice how the extra begin/end brace pair limit the scope of the loop and lLoop variables and that the loop limit is evaluated only once. This allows costly expressions such as strlen(). to be used as the loop limit because the evaluation takes place once, not on every loop iteration.

**Sample code that uses LOOP(), LLOOP() and ENDLOOP**

```
LOOP(10) {  
    printf( "%d\n", loop );  
} ENDLOOP  
LLOOP(10) {  
    printf( "%ld\n", lLoop );  
} ENDLOOP
```

You may be wondering why the C++ method of declaring the loop variable inside the for statement isn't used instead. This would allow you to get rid of the begin brace and ENDLOOP macro.

**C++ loop code, with variable declaration problems**

```
for (int loop=0; loop<10; ++loop) {  
    ...  
}  
...  
for (int loop=0; loop<10; ++loop) {  
    ...  
}
```

Unfortunately, the C++ method defines the loop variable from the definition point until the end of the current scope (but this is changing in C++). In

other words, the loop variable would now be known outside the scope of the loop and using two LOOP()'s in the same scope would end up declaring the looping variable twice, which cannot be done in the same scope.

### 2.2.6 NUMSTATICELS() Macro

Provided you have an array in which the number of elements in the array is known to the compiler, write a macro NUMSTATICELS() (number of static elements) that given only the array name returns the number of elements in the array. Consider the following example.

#### **NUMSTATICELS() desired behavior example**

```
#include <stdio.h>
#define NUMSTATICELS(pArray) (determine pArray array size)
int main(void)
{
    long alNums[100];
    printf( "array size=%d\n", NUMSTATICELS(alNums) );
    return 0;

} /* main */
```

#### **Desired output**

```
array size=100
```

How would you write NUMSTATICELS() so that the above example works? The solution is to write NUMSTATICELS() as follows.

#### **NUMSTATICELS() macro**

```
#define NUMSTATICELS(pArray) (sizeof(pArray)/sizeof(*pArray))
```

This macro works because sizeof(pArray) is the size in bytes of the entire array and sizeof(\*pArray) is the size in bytes of one element in the array. Dividing gives the maximum number of elements possible in the array. NUMSTATICELS() does not need to work on a pointer to a dynamically

allocated array.

So, in the above example, if we assume that `sizeof(long)` is 4, `sizeof(alNums)` is 400 and `sizeof(*alNums)` is 4. Dividing gives the desired output of 100.

### 2.2.7 Preprocessor Operators

Ask any C programmer what the preprocessor is used for and you hear things like macros, including header files, conditional compilation, and so on. These are obvious and useful features. However, ask the same C programmer what preprocessor operators are and you may get a blank stare.

The two most useful preprocessor operators are the stringizing operator (`#`) and the token pasting operator (`##`). Both of these operators are usually used in the context of a `#define` directive. The token pasting operator is at the heart of the class methodology introduced in [Chapter 4](#).

Stringizing operator (`#`). This operator causes a macro argument to be enclosed in double quotation marks. The proper syntax is `#operand`.

```
Stringizing operator example  
#define STRING(x) #x
```

Therefore, `STRING(hello)` yields `"hello"`.

In some older preprocessors, you accomplish stringizing of `x` in a macro (i.e., `#x`) directly by enclosing the macro argument `x` in quotes (i.e., `"x"`). Try this technique if you do not have a standard C compiler that allows stringizing.

Token pasting operator (`##`). This operator takes two tokens, one on each side of the `##` and merges them into one token. The proper syntax is `token1##token2`. This is useful when one or both tokens are expanded macro arguments. Consider the following example.

---

#### **Token pasting operator example**

```
CSCHAR szPE7008[]="function ptr";
CSCHAR szPE700A[]="string ptr";
CSCHAR szPE7060[]="coords";
...
#define EV(n) {0x##n,szPE##n}
static struct {
    WORD wError;
    LPSTR lpError;
} BASEDCS ErrorToTextMap[]={ EV(7008), EV(700A), EV(7060)
};
```

In this example, EV(7008) expands to {0x7008,szPE7008}, which initializes the wError and lpError elements of ErrorToTextMap (which happens to be in the code segment). For me, the EV() macro makes this code shorter and easier to understand.

### **2.2.8 Token Pasting in a Reiser Preprocessor**

In some old preprocessors that do not support the token pasting operator, it is still possible to perform token pasting provided the preprocessor follows the Reiser model. This is done by replacing ## with /\*\*/. This works because the comment /\*\*/ is removed and replaced with nothing, effectively pasting the tokens on either side of /\*\*/.

This technique does not work in newer standard C compilers that replace comments with a single space character. However, newer standard C compilers have the token pasting operator, so this technique is not needed.

### **2.2.9 Preprocessor Commands Containing Preprocessor Commands**

Have you ever wanted to write a macro that referred to another preprocessing directive? Consider the following example.

```
Optimize On/Off macros, which do not work
#define OPTIMIZEOFF #pragma optimize("",off)
#define OPTIMIZEON #pragma optimize("",on)
```



The OPTIMIZEOFF and OPTIMIZEON macros attempt to abstract out how optimizations are turned off and turned on during a compilation. The problem with these macros is that they are trying to perform another preprocessor directive, which is impossible with any standard C preprocessor. However, this does not mean that it cannot be done.

The solution to this problem is to run the source through the preprocessor twice during the compile instead of just once. Most compilers allow you to run only the preprocessing pass of their compiler and redirect the output to a file. If this output file is then run back through the compiler, the optimize macros work!

**Testing extra preprocessor pass in Microsoft C8 for C code**

```
cl -P test.c
```

```
cl -Tctest.i
```

**Testing extra preprocessor pass in Microsoft C8 for C++ code**

```
cl -P test.cpp
```

```
cl -Tpctest.i
```

## 2.3 Programming Puzzles

This section is designed to get you thinking. Some of the puzzles presented here have real programming value, while others have no programming value whatsoever. The point of these exercises is to get you thinking in a new light about things you already know about. If you want to skip this section, go to [§2.4](#)

### 2.3.1 ISPOWER2() Macro

Try to spend some time coming up with a macro that returns one if the macro argument is a power of two, or returns zero if the macro argument is not a power of two. The solution is as follows.



```
Macro that determines if a number is a power of two  
#define ISPOWER2(x)  (!((x)&((x)-1)))
```

This ISPOWER2() macro works for any number greater than zero because any number that is a power of two has the binary form "100...0", namely, a binary one followed by any number of binary zeros. Subtracting one from this number changes the leading binary one to zero and all trailing binary zeros to binary ones. Therefore, ANDing with the original number produces zero. Any number that is not a power of two has at least one bit in common with that number minus one. Therefore ANDing produces a non-zero number. Since this is just the opposite of the desired return value, the value is NOTed to get the desired result.

For example, the number 16 in binary is 00010000. Subtracting 1 from 16 is 15, which in binary is 00001111. So, ANDing 00010000 (16) with 00001111 (15) yields 00000000 (0) and NOTing gives 1. This tells us the number 16 is a power of two.

As another example, the number 100 in binary is 01100100. Subtracting 1 from 100 is 99, which in binary is 01100011. So, ANDing 01100100 (100) with 01100011 (99) yields 01100000 (96) and NOTing gives zero. This tells us that the number 100 is not a power of two.

### **2.3.2 Integer Math May be Faster**

Let us say you have an integer x and an integer y and you want x to contain 45 percent of the value contained in y. One obvious solution is to use `x = (int)(0.45*y);`. While this works, there is a shortcut that avoids floating point math and uses only integer arithmetic. 0.45 is just 45/100 or 9/20. So why not instead use `x = (int)(9L*y/20);`? The only disadvantage of this technique is that you need to watch out for overflows. Play around with this technique with some small numbers by hand until you get a good feel for what is going on.

### **2.3.3 Swap Two Variables without a Third Variable**

This is an old assembly language trick that can also be used in C. Given two

assembly language registers, how can you swap the contents of the registers without using a third register or memory location (and, of course, not using a swap instruction if the assembly you are familiar with has such an instruction). The solution is as follows.

```
Swapping integers x, y without a third integer using C  
1.  x ^= y;  
2.  y ^= x;  
3.  x ^= y;
```

After step 1, x contains  $(x \oplus y)$ . After step 2, y contains  $(y \oplus (x \oplus y))$ , which is just x. Finally, after step 3, x contains  $((x \oplus y) \oplus x)$ , which is y. This is it!

The trick to this technique is realizing that any number XORed with itself is 0. XOR also has useful applications in GUI environments.

### 2.3.4 Smoother XORs in a Graphical User Interface

A standard technique used in Graphical User Interfaces is to invert part of the display screen (using XOR) while the user is resizing a window to show the user the new size of the window. XORing again to the same screen locations restores the screen image to its original form.

So, when the user starts to resize the window, the location is marked by XORing the window border. When the user moves a little bit, the same location is XORed to remove the highlight, the proposed border is sized to the new location and it is XORed onto the screen, showing its new location. This process works pretty well but it results in screen flicker. The entire inverted window border is constantly being placed down and removed in its entirety.

Let's analyze the process in abstract terms using regions. We have three components: the screen (SCRN), the old border region (OBR) and the new border region (NBR). The border is placed down by a  $SCRN \oplus OBR$  operation. As the user moves the mouse, the goal is to turn the screen

containing the old border region into a screen containing the new border region. (See Figure 2-3).

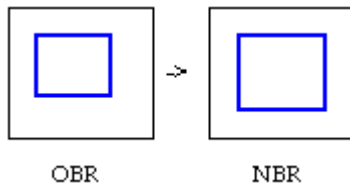


Figure 2-3: Moving a window border

The old way of implementing this is to remove the old border followed by placing down the new border. The old border is removed by a  $\text{SCRN} \wedge = \text{OBR}$  operation and the new border is placed down by a  $\text{SCRN} \wedge = \text{NBR}$  operation. In other words,  $\text{SCRN} = (\text{SCRN} \wedge \text{OBR}) \wedge \text{NBR}$ . (See Figure 2-4)

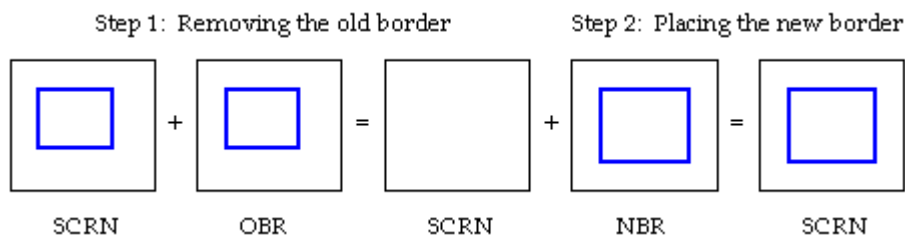


Figure 2-4: The standard way of moving a window border.

Does it really matter in what order you XOR? No, not at all, since XOR is associative. So why not instead do  $\text{SCRN} = \text{SCRN} \wedge (\text{OBR} \wedge \text{NBR})$ ? The screen now updates without even a hint of flicker! This is because you are finding the difference between the old border region and the new border region (i.e., XOR) and XORing that difference onto the screen. The result is that the old border is erased and that the new border is now visible. (See Figure 2-5).

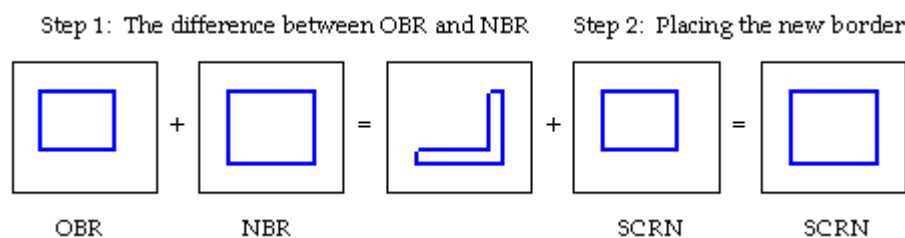



Figure 2-5: A better way to move a window border

The key to this magic is having region support provided to you by the environment you are working on.

 For Microsoft Windows, the functions of interest are `CreateRectRgn()`, `SetRectRgn()` and `CombineRgn()`.

### **2.3.5 Is $\sim 0$ More Portable Than `0xFFFF`**

How do you fill an integer variable so that all bits are turned on, but in a portable, data type size independent manner? Do you use `-1`? No, because this assumes a two's complement machine. What if you are on a one's complement machine? Do you use `0xFFFF`, `0xFFFFFFFF` or whatever? No, since this assumes that there are a particular number of bits in an integer.

The only thing you can be sure of in every machine is that zero is represented as all bits turned off. This is the key. There is a built-in C operator to flip all the bits and it is the one's complement  $\sim$  operator. Therefore,  $\sim 0$  fills an integer value with all ones in a portable manner.

This technique also works well to strip off the lower bits of a number in a portable manner. For example,  $\sim 7$  has all the bits in the resulting number turned on except for the lower three bits. Therefore, `wSize &  $\sim 7$`  forces the lower three bits of `wSize` to zero.

### **2.3.6 Does `y = -x`; Always Work?**

You may be wondering if I wrote that statement correctly. I did. Do you know when `y = -x` fails? It fails on a two's complement machine when `x` equals the smallest negative number. Consider a 16-bit two's complement number where the smallest negative number is `-32768`. So, if it fails, what is the result of negative `-32768`? It is `-32768`. The valid range of numbers on a two's complement machine for a short integer is `-32768` to `32767`. There is one more negative number than there are positive numbers.

### **2.3.7 One's and Two's Complement Numbers**

In one's complement notation, the negative of a number is obtained by

inverting all the bits in the number. For short (16-bit) integers, this results in a range from -32767 to 32767. So what happened to the extra number? Well, there are now two representations for zero. Namely a negative zero (all bits on) and a positive zero (all bits off). One's complement works, but it requires special case hardware to make sure everything works. One day someone came up with the bright idea of two's complement, which is simply one's complement plus one. It eliminates the two representations for zero and eliminates the special case hardware.

```
Two's complement negation as a macro  
#define NEGATE(x) (((x) ^ ~0) + 1)
```

So why can two's complement be implemented so efficiently? For simplicity, let's assume a three-bit unsigned integer. The integer can take on the values from 0 to 7. Any arithmetic on the numbers are modulo 8 (remainder upon division by 8). As an example, adding 1 to 7 is 0 and adding 3 to 7 is 2. You can verify this by going to the first number (e.g., 7) on the base eight number line and moving right the number of times specified by the second number (e.g., 3).

```
Base eight number line  
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 ...
```

You will quickly realize that adding 7 to any number is like subtracting 1, adding 6 is like subtracting 2, and so on. So what if you simply label 4 through 7 slightly differently.

```
Base eight number line as signed numbers  
0 1 2 3 -4 -3 -2 -1 0 1 2 3 -4 -3 -2 -1 ...
```

Half the numbers are now negative and the other half are for zero and positive numbers. So adding negative 1 (i.e., 7) to any number really does work. That is it! This two's complement technique can be extended to any

bit size.

## 2.4 Microsoft Windows

One of the first things one learns about the Windows environment is that it allows for true code sharing. An example of this is running an application, such as Notepad, more than once. A new data segment is created for each new instance of the application, code segments and resources being shared between the two instances. This is why there are such things as "handle to a module" and "handle to an instance." The module handle basically refers to the executable file, the common part per instance, and the instance handle refers to the application data segment, the unique part per instance.

### 2.4.1 Dynamic Link Libraries (DLLs)

A more powerful mechanism of sharing both code and data is provided through the use of Dynamic Link Libraries or DLLs. In fact, most of the Windows environment is a set of DLLs, namely the USER, KERNEL and GDI DLLs. For DLLs, no matter how many applications use or reference them, only one set of code is loaded and only one data segment is loaded (when using the preferred medium memory model).

Writing DLLs allows you to write code that is truly sharable among applications and promotes the "write-it-once" attitude. After all you have done to learn Windows, why not write the majority of your code as a DLL so that you now have a library of code waiting to be used by your next killer application. You may argue that there is no need for this since all you have to do is copy the code, but what you have gained is code maintainability. Let's say you rewrite some of the shared code. With a DLL you change it once. With multiple copies of the code you have to change it multiple times.

### 2.4.2 Special DLL DS!=SS Concerns

DLL programming is one of the least understood concepts of Windows programming because it is considered "Advanced Programming" and is always one of the last chapters of any Windows book. DLL programming

is also more difficult than programming for an application due to "DS!=SS" (data segment does not equal stack segment) concerns.

In a medium model Windows program you typically have multiple code segments and one data segment. The stack for the application is contained within the application data segment so that near pointers can access both data segment data and stack segment data. It then makes sense to set DS==SS (data segment equals stack segment). When a DLL is called, the data segment is changed but not the stack segment. You now have a "DS!=SS" situation. This allows the DLL function to access parameters passed to the function (on the stack) and private data (in the DLL data segment). However, a near pointer can access only the data segment in a DLL, not the stack segment. This causes problems for a fair number of useful C library functions. From experience I would stay away from C library calls in a DLL unless you have a good reason to use them. Instead, code the functionality yourself.



### **2.4.3 Real-Mode Windows**

Windows 1.x and 2.x were real-mode operating systems. They did not use the protected-mode architecture of the Intel CPU. This all changed with the introduction of Windows 3.0. It supports real-mode, standard-mode (a.k.a. 286 mode) and enhanced-mode (a.k.a. 386 mode). With the introduction of Windows 3.1, real-mode support has been removed.

Have you ever wondered how Windows running in protected-mode could continue to run old Windows applications that were designed for real-mode? That this even works is a tribute to the original designers of Windows 1.0! The question should be rephrased as follows: Can you believe that Microsoft got a protected-mode architecture to run under the real-mode of the CPU?

I want to give you an inside look into one of the more interesting features of Windows and how it was implemented under real-mode. The feature is discardable code segments.

An application under Windows 2.x is usually composed of many code



segments. Most of the code segments were movable and discardable. In other words, the code could be moved around in memory by the operating system as needed and if memory became tight, the code segment could be discarded from memory to make room for other things.

As an example, let's say you have function A in segment A that calls function B in segment B. There is now a far return address on the stack. Now let's assume that segment A is discarded. Under a protected-mode operating system, returning to function A would fault to the operating system and segment A would simply be loaded back into memory. What about real-mode? Returning to function A cannot cause a CPU fault.

How would you solve this problem?

The solution is rather ingenious and requires compiler support to be fully implemented. Suppose that a segment is moved in real-mode memory. Since the segment address has changed, walk the stack and patch those return addresses on the stack that have the old segment address with the new segment address. If the segment is discarded, walk the stack and patch the address with an address into the operating system that reloads the segment.

Walking the stack under the Intel segmented architecture is complicated by the fact that the addresses can either be near (16 bit) or far (32 bit). So when walking the stack the operating system has no way of knowing if it should expect a near or far address. It cares only about the far addresses because near addresses do not have to be patched.

The solution has to do with the stack frame that is built for all functions and the fact that the Intel stack segment is word aligned.

Within each stack frame is a pointer to the previous stack frame. Since this pointer is word aligned, the lower bit is always zero. If you were to use this bit as a near/far indicator, you could then walk the stack with no problems. The compiler generates the code that correctly sets this bit to indicate the near or far nature of the function.

This technique was used by real-mode Windows compilers but is now no longer used since Windows is a protected-mode only operating system.



#### 2.4.4 Why MakeProcInstance() Was a Design Flaw

After seeing how good a job Microsoft did with designing Windows to work under real- and protected-mode you may be amazed to learn how another problem was solved.

When an application such as Notepad is run multiple times, only one set of code ever gets loaded because code is shared under the Windows operating system. However, each Notepad does get its own copy of its data segments. Because Windows is event driven, there are many occasions where Windows calls your code. When it calls your code, how does Windows bind to the correct data segment?

The solution the Windows designers came up with is that instead of providing the address of the callback directly to Windows, you instead pass the address of some thunk code that binds to the correct data segment and then calls the true callback. This thunk code is created using `MakeProcInstance()`.

A thunk is usually just a small piece of code that gets executed just before a real function gets executed. The purpose of a thunk is usually to bind something to the function. In the case of `MakeProcInstance()`, this something is the correct data segment.

##### **MakeProcInstance() prototype**

```
FARPROC MakeProcInstance( FARPROC lpProc, HINSTANCE hInst );
```

However, as it turns out, the correct data segment value used by the thunk code is already in the SS (stack segment) CPU register because an application's stack is contained in its data segment (i.e.,  $DS==SS$ ). This eliminates the need for the thunking code and hence `MakeProcInstance()` because the correct data segment to use is always in the SS register.

All compilers for the Windows environment now have compiler switches that bind to the correct data segment from the value in the SS register. MakeProcInstance() never needs to be used again. Under Microsoft C8, the command line options for this is /GA (for applications).

This technique of eliminating the need for MakeProcInstance() was originally developed by the brilliant Windows programmer Michael Geary and distributed in his FixDS utility. For an interesting discussion about Geary and the Adobe Type Manager, refer to "The Geary Incident" in Undocumented Windows.

### 2.4.5 Optimizing the Compiler Options

The Microsoft C8 compiler assumes the worst when selecting default command line options. The compiler defaults to options that will work in all environments. However, the default options are not always the best options and not optimizing them will add execution overhead to your program.

Instruction sets. For example, since Windows now requires at least a 286 processor (i.e., protected-mode), you can use the /G2 command line option, which tells the compiler to generate instruction sequences using the 286 instruction set. If you know that your program will be run only on 386's or better, you can even use the /G3 option, which tells the compiler to generate code using the 386 instruction set.

Prolog/Epilog code generation. Due to historical reasons in Windows using the real-mode architecture of the Intel processor, far functions required special treatment by the compiler. However, if your application is a protected-mode Windows application (and every one is today), you can optimize how the compiler generates code for these far functions. For applications, use the /GA command line option and explicitly place the `__export` keyword on any callback functions. For DLLs, use the /GD command line option and explicitly place the `__export` keyword on any callback or API functions.

Pascal calling convention. To reduce the amount of code generated to

support function argument passing, it is recommended that you use the Pascal calling convention. The /Gc command line option tells the compiler that all functions should default to the Pascal calling convention. See the [Pascal calling convention §2.1.11](#) and how it compares to the default [C calling convention §2.1.11](#).

Remove stack-check calls. It is recommended that you compile your Windows program with stack probe checks turned off. This is done through the /Gs command line option.



## **2.4.6 Dynamic Dynamic Linking**

Dynamic linking is at the core of the Windows operating system. It is a mechanism for resolving references to operating system components at load-time. For example, if your program calls the Windows PolyLine() function, your program does not contain the PolyLine() code. Instead, it contains a call to code that does not exist in your program. When the program is loaded into memory, the operating system patches your code so that it calls the PolyLine() code contained in the operating system. This is dynamic linking.

Dynamic dynamic linking allows you, the programmer, to link to a function at run-time. This is done through the Windows LoadLibrary(), GetProcAddress() and FreeLibrary() functions.

One good reason for using dynamic dynamic linking is that it allows a program to optionally use advanced features of the operating system while still allowing the program to be run on minimally configured systems. Suppose you write a simple terminal emulator that uses either an asynchronous port or a network port. By linking to the network library, your program cannot be run on machines that have only an asynchronous port and not a network connection because the program requires the network library to be present in order to be run. However, by using dynamic dynamic linking to the network library, your program can attempt to bind to the network library only if the user has selected an option that requires the network library to be present.

## 2.4.7 Messages

Windows is a message-based system. Each application running in the system has a message queue from which it is responsible for removing and dispatching messages.

It is important to realize that not all Windows messages are alike. Some messages are telling your program that an event has already happened (notification messages). Other messages are requesting your program to perform some action (action messages). This distinction is subtle and yet important to understand because knowing the difference will allow you to code simple solutions to what seem to be complex problems.

Notification messages. Receiving the `WM_MOUSEMOVE` message is an example of a notification message. Whenever the mouse is moved on the screen, some window will receive this message. Ignoring this message has no effect. The mouse pointer will continue to move on the screen.

Action messages. Receiving the `WM_COMMAND` message is an example of an action message. It is usually sent to your program in response to the user's selecting a menu item. Ignoring this message would be serious because your program is supposed to perform an action based upon this message.

Suppose you have a dialog box that contains several edit controls. The tendency of users of this dialog box is to type text into the first edit control and then press the enter key to advance to the next edit control. The problem with pressing the enter key is that it is the same as pressing the OK button, which exits the dialog box! So how can you allow the enter key to be pressed to advance to the next edit control?

The first inclination of a lot of programmers is to attempt to use subclassing to solve this problem. However, there is a much simpler solution. Pressing the enter key results in the dialog box manager sending an `IDOK WM_COMMAND` message to the dialog box. This message is an action message, not a notification message. The solution is to check in the `IDOK` processing code if the focus is on the OK button. If so, the

dialog may be exited; otherwise advance the focus to the next edit control (and do not exit the dialog).

Trapping action messages is easier than subclassing in some cases.

Another good example involves moving windows. When the user moves the mouse over the caption area of a window and drags, the window is moved by Windows. Suppose you want this same moving behavior when the user clicks and drags in the window. At first glance it would appear that you would have to write a lot of code. The solution involves realizing how the WM\_NCHITTEST action message works. This message is sent by the Windows internals asking the window to perform hit testing. Windows wants to know what part of the window (caption bar, left border, right border, client area, etc.) a certain point is in. The following code is the solution to the problem.

**Changing a window's client area to behave like the caption area**

```
case WM_NCHITTEST: {
    LRESULT lRet=DefWindowProc(hWnd, message, wParam,
lParam);
    return ((lRet==HTCLIENT)?HTCAPTION:lRet);
    break;
}
```

Placing this code in the window procedure of the appropriate window solves the problem. Since WM\_NCHITTEST is an action message, we first call the default window procedure, DefWindowProc(), to perform the standard hit testing. Next we return the result of the hit test, but change the client area (HTCLIENT) to look like the caption area (HTCAPTION).

## 2.5 Chapter Summary

- Before you can efficiently institute new programming methodologies that help reduce bugs in your programs, you need to fully understand

your programming environment. Try to become an expert in your environment.

- Study your environment and learn as much about it as you can. The learning process never stops. Even if you have been programming in C for many years, you will still learn new nuances about C all the time.
- Your goal should be to become an expert in your programming environment. While becoming an expert is not a prerequisite to developing programming methodologies, it does help you develop more advanced methodologies.

# Chapter 5: A New Heap Manager [Writing Bug-Free C Code](#)

[5.1 The Traditional Heap Manager](#)

[5.2 A New Heap Manager Specification](#)

[5.3 An Interface for Strings](#)

[5.4 An Interface for Arrays](#)

[5.5 Detecting Storage Leaks](#)

[5.6 Windows Memory Model Issues](#)

[5.7 Chapter Summary](#)

This chapter introduces a new heap manager interface that checks for common mistakes on the part of the programmer allocating and freeing memory and also fulfills the requirements of the class methodology introduced in [Chapter 4](#). All of the code introduced in this chapter can also be found in one convenient location in the [Code Listings Appendix](#).

## 5.1 The Traditional Heap Manager

A good friend of mine was visiting one day when we happened to start talking about programming techniques. He had landed a job with a well-known, large computer company. I explained to him the replacement heap manager that I had come up with and the reasons why I felt that it was necessary. I then questioned my friend on the programming guidelines concerning memory deallocations that were in place at his company.

As it turns out, there were none. The standard C library routines were it. Throughout the code there were direct calls to the memory manager to allocate and free memory. I then asked what was done when an invalid memory pointer was inadvertently passed to `free()`. Nothing was done. It was assumed that memory pointers passed to `free()` were always valid. Whenever a memory management bug occurred, it would be tracked down and fixed.

Tracking down memory management bugs as they occur violates the principle of solving the problem that caused the bug and not just the manifestation of the bug.

### 5.1.1 The Interface



The standard C library routines for allocating and freeing memory are `malloc()` and `free()`.

**Standard C library memory function prototypes**

```
void *malloc ( size_t size );  
void free   ( void *memblock );
```

The `malloc()` function takes one argument, the size of the memory object to allocate, in bytes. It returns a void pointer to the allocated block of memory or NULL if there is insufficient memory or if some error exists. The memory is guaranteed to be properly aligned for any type of object.

The `free()` function takes one argument, a void memory pointer that was previously allocated through a `malloc()` call. The `free()` function has no return value. If a NULL pointer is passed to `free()`, the call is ignored. If an invalid memory pointer is passed to `free()`, the behavior of `free()` is undefined.

While there are several more standard functions for manipulating memory, for the sake of discussion, we are interested only in `malloc()` and `free()`.

### 5.1.2 The Problem

The single biggest problem with the standard C library interface to memory is that it assumes the programmer never makes a mistake in calling the memory management functions. Consider the following.

**C program, with memory management problems**

```
#include <stdlib.h>  
  
int main(void)  
{  
    void *pMem=malloc(100);  
    free(pMem);  
    free(pMem);  
    return 0;
```

```
} /* main */
```

Using `free()` on a memory pointer that has already been passed to `free()` is a common error in many large C programs. How `free()` behaves in such a case is undefined. On some systems, the run-time library may tell you that you have made a mistake and terminate your program. Other systems will continue, even though the validity of the application's heap is in question.



This is what happens in Microsoft C8. The sample code above runs just fine, producing no error messages. In Microsoft C8, the documentation for `free()` states the following:

*Attempting to free an invalid pointer may affect subsequent allocation and cause errors. An invalid pointer is one not allocated with the appropriate call.*

C was designed to be portable and produce small, efficient code. To accomplish this feat so successfully, a lot of small but important decisions have been left to be decided by the particular implementor. The designers of C wanted to give as much leeway as possible to each particular implementation.

The C language provides a minimal, efficient framework.

It is up to the programmer to build upon this framework.



### 5.1.3 Windows Issues

A problem for many Windows programmers is what memory interface to use. The standard C library provides one interface (`malloc()` and `free()`) and the Windows API provides two interfaces based upon handles to memory objects. The first Windows API is a local memory interface (`LocalAlloc()`, `LocalLock()`, `LocalUnlock()` and `LocalFree()`) in which the total of all local objects must be less than 64K. The second Windows API is a global

memory interface (GlobalAlloc(), GlobalLock(), GlobalUnlock() and GlobalFree()), which allows for a large number of varied-sized objects to be allocated.

The reasons for the handle-based interface used by Windows is largely historical and can be almost totally ignored today in favor of using the C library interface. When Windows supported the real-mode of the Intel processor, the handle-based alloc/lock/unlock/free model allowed the Windows kernel to move memory around to avoid fragmentation. However, now that Windows supports only the protected-mode of the Intel processor, the handle-based model is no longer needed because the processor is capable of memory management tasks, even on locked memory objects.

The only time that you need to deal with the handle-based memory interface is when you are dealing with those Windows API calls that use or return a memory object based upon handles. SetClipboardData() and GetClipboardData() are two examples of Windows APIs that still use the handle-based interface.

## **5.2 A New Heap Manager Specification**

A clear set of goals is needed before a new heap manager specification can be designed. The job is finished when the goals have been accomplished.

### **5.2.1 Flat and Segmented Architectures**

*Flat memory model.* For programmers using machine architectures that are based upon a flat (non-segmented) memory model, the choice for a heap manager interface is straightforward. There are no choices about which pointer size to use. Only one address size exists and it is usually 32 bits.

*Segmented architectures.* For programmers using machine architectures that are segmented, the choice for a heap manager interface is not at all straightforward. In fact, it is filled with a lot of choices. The primary reason for this is usually a concern for speed.

The following discussion centers around the segmented architecture used by Intel.

An address in a segmented architecture is composed of two parts. Part of the address is used to determine which segment is being used. The other part of the address is used to determine the byte within the segment. (See Figure 5-1).

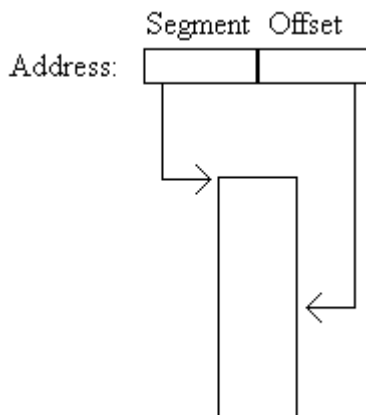


Figure 5-1: A segmented architecture address.

There are two primary ways an address can be specified. The first is by specifying the full segment and offset (far address). The second is by specifying only the offset (near address), where the segment is implied to be one of the segment registers contained in the CPU. In terms of execution speed, specifying only the offset part of an address is fastest.

You may be thinking that the choice of which addressing method to use is easy. Pick near addresses for speed. The problem here is that a segment can address only 64 kilobytes of memory (i.e., 65,536 bytes). If the total amount of memory that you need to allocate is less than 64K bytes, then great; use near addresses for your heap pointers. However, if the total amount of memory that you need to allocate is more than 64K bytes, then you are forced to use multiple segments and hence far addresses.

All compilers allow different memory model programs to be built because of the different addressing possibilities: small, for when code and data together are less than 64K; compact, for code less than 64K and data greater

than 64K; medium, for code greater than 64K and data less than 64K; large, for code and data greater than 64K.

This situation is complicated by the fact that all C compilers allow pointers to be tagged on a case-by-case basis as either near or far. This is called mixed-model programming.

All segmented and non-segmented issues can be isolated through the use of a set of macros that act as an interface to memory. A heap manager interface can be tailored to the specific needs of your environment and program. There is no need to come up with a super memory management routine that allows megabytes of memory to be allocated if you are writing small utility programs requiring only a few kilobytes of memory.

I feel that segmented architectures have gotten a bum rap. It is true that a 64K segment is a bit limiting, but this is only one implementation. It can be implemented in a better way. With 64-bit architectures coming down the road, there are a lot of possibilities! The biggest advantage of a segmented architecture is that every segment is protected from every other because all memory references are checked for validity in two very special ways. First, the segment value must be a valid segment. Second, the offset must be within the limits of the segment. If either check fails, the memory access generates a protection violation. Memory overwrites on a heap object are detected by the hardware, even an overwrite of 1 byte. This feature is great for debugging.

### **5.2.2 Requirements**

The requirements of the new heap manager interface are as follows.

*Invalid memory pointers are detected and reported.* First and foremost on the list is to plug the gap left by the standard C malloc() and free() functions. Invalid memory pointers passed to the memory deallocation routine are automatically detected and reported.

*Memory allocation/deallocation performance must not be adversely affected.* The last thing we want to do is come up with a set of requirements

that are expensive to implement in terms of execution time. All objects in our class methodology are dynamically allocated and freed and we want to be careful not to adversely impact the performance of the object system.

*Support for run-time type checking.* All objects in the heap must have a header containing type information preceding each object. The [object system \(Chapter 4\)](#) requires a specific format for the two data items immediately preceding the object.

*Memory is zero initialized.* Memory that is allocated and freed is zero initialized. Memory is zero initialized for convenience. Knowing that all items of an object, after a NEWOBJ(), are bit initialized to zero is a nice feature to have. Memory is zero initialized when being deallocated for any incorrect references to the object after the object is deallocated. This is highly unlikely, however, because the FREE() macro sets the object pointer to NULL.

*Memory allocations do not fail.* This requirement makes sense only in a virtual memory environment. If a call is made to the heap management memory allocator, the call returns successfully. In the case of an error, it does not return at all and reports the error. This prevents having to place failure checks everywhere in the code. In the majority of applications, the maximum amount of memory that can be allocated by the application is well within the virtual memory limits for an individual program. In these cases, this requirement makes a lot of sense.

For the minority of applications whose memory allocations may not fit into the virtual address space limits, the memory allocator should return a failure status (NULL). Error checks must then be made throughout the code.

In the spirit of the design of C, it is up to you to decide how to implement this requirement.

*Memory overwrites past the end of the object are detected and reported.* The heap manager provides memory space for any type of object. This includes true objects in the object system as well as strings and whatever else is needed. Writing past the end of a dynamically allocated string should

be detected and reported. Accidentally writing past the end of a class object is highly unlikely.

*Storage leak detection.* As a program executes, it allocates and frees memory. At program termination, if there is any allocated memory remaining in the heap, that memory is considered to be a storage leak. Any storage leaks should be reported.

*Filename and line number tags.* All heap objects should be tagged with the filename and line number at which the object was created. This information is useful for producing heap dumps and is required for producing useful information on storage leaks.

*Symbolic heap dumps.* The heap manager must know how many objects there are in the heap and must be able to walk the heap, producing useful information about all objects in the heap. The heap dumps are primarily for tracking down the cause of storage leaks.

*Alignment preservation.* Any special data alignment requirements of the CPU must be met. Most RISC architectures require that data items be aligned on 2, 4, 8 or even 16-byte boundaries. Even on architectures with no absolute alignment requirements, it is usually more efficient to have aligned data items because the hardware expends extra CPU cycles to align unaligned data.

*Be as portable as possible.* The ideal interface would be easy to port to any system. While this is possible, the execution time on the varied platforms would probably be considerable. For this reason, two layers should be used. The first is a set of macros that present to the programmer a logical, high-level view of memory. An example of a macro like this is the NEWOBJ() macro. The second layer is the heap manager function call specification, which is used by the macros.

The only way to allocate and deallocate memory is through this set of macros. The macros, in turn, call the actual heap manager functions. If you port to another platform, simply tailor the heap manager to the particular environment, change the macros that call the new heap manager and

recompile.

In practice, I have found that this technique works well.

Freeing the NULL pointer is OK. Calling the memory deallocator with the NULL pointer is allowed and the call is simply ignored. In practice, I have found this feature to be useful because it prevents having to bracket every memory deallocation in an if statement, checking to see if the pointer is non-NULL.

### 5.2.3 The Interface

The heap manager interface that I recommend is based upon the 32-bit model. Whether the architecture of the machine is segmented or not is not an issue (except for its performance impact). Users of the heap interface do not know and do not need to know the true nature of the 32-bit pointers being returned by the heap interface. The returned address could be composed of both segment and offset or it could be a flat memory pointer. The calling code works the same in either case.

To support [run-time type checking §4.4](#), the memory allocator must be passed the address of a class descriptor. If the object being allocated is not a class object, NULL should be passed as the class descriptor address.

To support storage leak detection and symbolic heap dumps, a source filename and line number must be passed to the memory allocator. It is assumed that the source filename address is always accessible.

All the other heap manager requirements can be met within the heap manager. With this information, the prototypes to the heap manager interface can be written as follows.

#### **The heap manager interface**

```
EXTERNC LPVOID APIENTRY FmNew      ( SIZE_T, LPCLASSDESC,  
LPSTR, int);  
EXTERNC LPVOID APIENTRY FmFree     ( LPVOID );  
EXTERNC LPVOID APIENTRY FmRealloc  ( LPVOID, SIZE_T, LPSTR, int  
);
```



```
EXTERNC LPVOID APIENTRY FmStrDup    ( LPSTR, LPSTR, int );  
EXTERNC void    APIENTRY FmWalkHeap ( void );  
EXTERNC BOOL    APIENTRY FmIsPtrOk  ( LPVOID );
```

FmNew() (far memory new) takes a memory size, a class descriptor pointer, a source filename and line number as arguments. It returns a pointer to the allocated memory.

FmFree() (far memory free) takes a pointer to a previously allocated block of memory as an argument and returns the NULL pointer. Passing NULL to FmFree() is allowed and no action is taken in this case.

FmRealloc() (far memory realloc) is used to reallocate the size of a block of memory.

FmStrDup() (far memory string dup) is used to duplicate a string. It is intended to replace strdup().

FmWalkHeap() (far memory heap walk) walks the heap displaying every item in the heap.

FmIsPtrOk() (far memory address validation) is used to validate that the given pointer is a valid address.

#### **5.2.4 The Design**

The new heap manager takes the classic approach of providing a wrapper around a system's current heap manager. The wrapper includes both code and data. (See Figure 5-2).

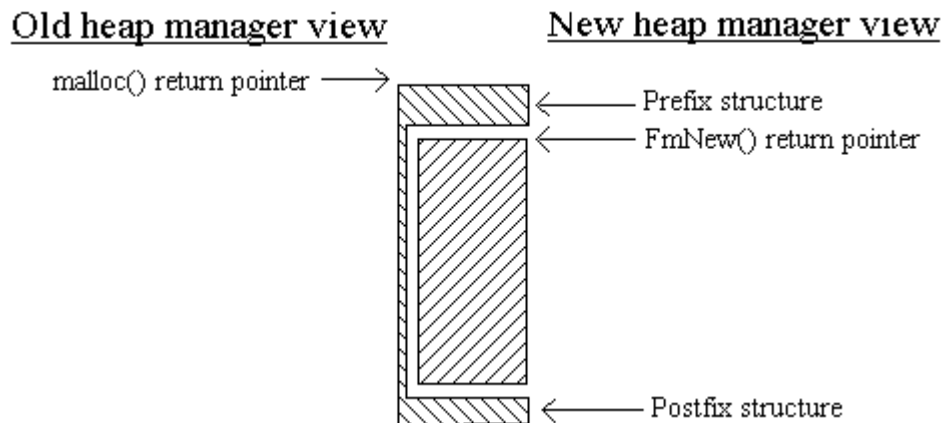


Figure 5-2: Viewing an object in the heap.

The code wrapper is in the form of the new `FmNew()` and `FmFree()` function calls that sit upon a system's existing `malloc()` and `free()` calls.

The data wrapper is in the form of a prefix structure that sits in front of a block of allocated memory and a postfix structure that sits after the block of memory.

The new heap manager is the only code that interfaces with the old heap manager. This is important since in order to run-time type check objects in the heap, the run-time type checking code assumes the new heap manager view, one in which heap objects are prefixed with type checking information.

The new heap manager is the only code that interfaces with the old heap manager.

The prefix structure is a container for holding information that needs to be associated with the object that is allocated. It contains type information, source filename and line number information and whatever else is needed to meet the heap manager's requirements.

The postfix structure is for detecting memory overwrites past the end of an allocated object. While it is no substitute for hardware overwrite protection and does not catch all memory overwrites, it catches the majority of memory overwrites. Most memory overwrites are caused by overwriting the end of a string buffer. This type of overwrite is caught by the usage of a

postfix structure.

A way to walk the heap is still needed in order to provide symbolic heap dumps and to detect storage leaks. Some environments provide a mechanism for walking the heap, while others provide no such mechanism. Rather than making any assumptions about the environment that you are running under, heap walking support is provided by the new heap manager. The simplest way to implement this is through a doubly linked list. This requires a next and previous pointer stored in the prefix structure.

All other requirements previously specified are implementation details. The layout of the prefix and postfix structures can now be designed.

#### **The prefix and postfix structures**

```
/*--- Declare what LPPREFIX/LPPOSTFIX are ---*/
typedef struct tagPREFIX FAR*LPPREFIX;
typedef struct tagPOSTFIX FAR*LPPOSTFIX;

/*--- Prefix structure before every heap object---*/
typedef struct tagPREFIX {
    LPPREFIX lpPrev;           /* previous object in heap
*/
    LPPREFIX lpNext;          /* next object in heap
*/
    LPPOSTFIX lpPostfix;      /* ptr to postfix object
*/
    LPSTR lpFilename;         /* filename ptr or NULL
*/
    long lLineNumber;         /* line number or 0
*/
    LPVOID lpMem;             /* FmNew() ptr of object
*/
    LPClassDesc lpClassDesc;  /* class descriptor ptr or NULL
*/
} PREFIX;

/*--- Postfix structure after every heap object ---*/
typedef struct tagPOSTFIX {
    LPPREFIX lpPrefix;
} POSTFIX;
```

The prefix structure contains lpPrev and lpNext for maintaining a linked list of objects in the heap; lpPostfix, for pointing to the postfix structure after the heap object; lpFilename and lLineNumber, for source filename and line number support; lpMem and lpClassDesc, for run-time type checking support.

The postfix structure could contain anything since it is being used for memory overwrite detection. The contents of the structure are initialized when the heap object is created and verified when the heap object is destroyed. However, instead of filling in the structure with a constant value when the heap object is created and checking that constant when the heap object is destroyed, it is better to use a value that varies from heap object to heap object. A mechanism for achieving this result is to use a pointer back to the prefix object.

The only real outstanding issue is alignment preservation. Because malloc() returns a pointer that is correctly aligned, the prefix structure is properly aligned. The pointer that is returned from FmNew() immediately follows the prefix structure. You have two choices to make sure this pointer is aligned: Either align it in code, or make sure that the size of the prefix structure is a multiple of the alignment. The simplest technique is to make sure that the prefix structure is a multiple of the alignment. If it is not, pad the structure with dummy data items. In doing so, however, make sure that you pad at the beginning of the structure and not at the end. This is because the lpMem and lpClassDesc data items must be next to the allocated object. The prefix structure is currently correct for an alignment of sizeof(int). You can change the ALIGNMENT macro if sizeof(int) is incorrect for your environment. The CompilerAssert() macro should be used to guarantee the correct alignment.

```
Guaranteeing correct prefix structure alignment  
#define ALIGNMENT (sizeof(int))  
CompilerAssert (ISPOWER2 (ALIGNMENT)) ;  
CompilerAssert (! (sizeof (PREFIX) %ALIGNMENT)) ;
```

The ALIGNMENT define and the ISPOWER2 CompilerAssert are placed near the top of the source file. The PREFIX CompilerAssert is placed in the source file after the prefix structure has been declared.

The pointer returned from FmNew() is now properly aligned. To properly align the postfix structure, use the same trick as before, but this time in code. Simply take whatever size is passed into FmNew() and increase its size, if needed, to the nearest alignment boundary.

#### **Aligning a memory size, assumes no overflow**

```
#define DOALIGN(num) (( (num)+ALIGNMENT-1) &~ (ALIGNMENT-1) )
```

The DOALIGN() macro aligns a number up to the nearest ALIGNMENT boundary. It assumes that there is no overflow and that ALIGNMENT is a power of two (the reason for the ISPOWER2 CompilerAssert() above).

### **5.2.5 FmNew() Implementation**

#### **FmNew() function**

```
LPVOID APIENTRY FmNew( SIZET wSize, LPCLASSDESC lpClassDesc,
    LPSTR lpFile, int nLine )
{
    LPPREFIX lpPrefix;
    wSize = DOALIGN(wSize);
    lpPrefix=
(LPPREFIX)malloc(sizeof(PREFIX)+wSize+sizeof(POSTFIX));
    if (lpPrefix) {
        AddToLinkedList( lpPrefix );
        lpPrefix->lpPostfix = (LPPOSTFIX)((LPSTR)
(lpPrefix+1)+wSize);
        lpPrefix->lpPostfix->lpPrefix = lpPrefix;
        lpPrefix->lpFilename = lpFile;
        lpPrefix->lLineNumber = nLine;
        lpPrefix->lpMem = lpPrefix+1;
        lpPrefix->lpClassDesc = lpClassDesc;
        memset( lpPrefix->lpMem, 0, wSize );
    }
    else {
        AssertError;                /* Report out of memory error */
    }
}
```

```

    }
    return(lpPrefix ? lpPrefix+1 : NULL);
} /* FmNew */

```

The implementation of FmNew() is straightforward. The first thing to do is align wSize up to the nearest alignment boundary. A block of memory is then allocated, the size of which is big enough to hold the prefix structure, the user block of memory and the postfix structure.

The call is to malloc(), a 32-bit memory interface. The 32-bit memory allocator may be called by other names in other environments. Simply replace malloc() with a call that is appropriate to your environment.

If the memory allocation fails, the FmNew() code essentially exits with NULL. You may want to implement a policy where memory allocations must succeed and if they do not, the FmNew() function does not return to the calling code.

If the memory allocation succeeds, the first thing to do is add the memory block to the doubly linked list of memory blocks, by calling AddToLinkedList().

```

AddToLinkedList() function
static LPPREFIX lpHeapHead=NULL;

void LOCAL AddToLinkedList( LPPREFIX lpAdd )
{
    /*--- Add before current head of list ---*/
    if (lpHeapHead) {
        lpAdd->lpPrev = lpHeapHead->lpPrev;
        (lpAdd->lpPrev)->lpNext = lpAdd;
        lpAdd->lpNext = lpHeapHead;
        (lpAdd->lpNext)->lpPrev = lpAdd;
    }
    /*--- Else first node ---*/
    else {
        lpAdd->lpPrev = lpAdd;
        lpAdd->lpNext = lpAdd;
    }
}

```

```

    /*--- Make new item head of list ---*/
    lpHeapHead = lpAdd;

} /* AddToLinkedList */

```

Once added to the linked list, lpPostfix is filled in to point to the postfix structure. The lpPrefix data item within the postfix structure is then filled in. It simply points back to the prefix structure. Next, the lpFilename and lLineNumber data items are initialized from the arguments passed to FmNew(). Finally, the lpMem and lpClassDesc data items are initialized for purposes of run-time type checking.

Finally, the block of user memory is zero initialized by calling memset().

### 5.2.6 FmFree() Implementation

```

FmFree() function
LPVOID APIENTRY FmFree( LPVOID lpMem )
{
    if (VerifyHeapPointer(lpMem)) {
        LPPREFIX lpPrefix=(LPPREFIX)lpMem-1;
        SIZE_T wSize=(LPSTR) (lpPrefix->lpPostfix+1)-
(LPSTR) lpPrefix;
        RemoveFromLinkedList( lpPrefix );
        memset( lpPrefix, 0, wSize );
        free(lpPrefix);
    }
    return (NULL);
} /* FmFree */

```

The FmFree() implementation is straightforward, provided that an effective VerifyHeapPointer() can be written. This function has all the implementation problems that run-time type checking has. VerifyHeapPointer(), in order to be completely robust, must be tailored to a specific machine architecture.

Provided that the memory pointer passed to FmFree() is indeed a valid heap pointer, VerifyHeapPointer() allows the body of FmFree() to execute. The

first thing to do is calculate a pointer to the prefix structure. The size of the object is calculated next and assigned to `wSize`. The heap object pointed to is removed by calling `RemoveFromLinkedList()`.

#### **RemoveFromLinkedList() function**

```
void LOCAL RemoveFromLinkedList( LPPREFIX lpRemove )
{
    /*--- Remove from doubly linked list ---*/
    (lpRemove->lpPrev)->lpNext = lpRemove->lpNext;
    (lpRemove->lpNext)->lpPrev = lpRemove->lpPrev;
    /*--- Possibly correct head pointer ---*/
    if (lpRemove==lpHeapHead) {
        lpHeapHead = ((lpRemove->lpNext==lpRemove) ? NULL :
            lpRemove->lpNext);
    }
}

/* RemoveFromLinkedList */
```

Once the memory object has been removed from the doubly linked list, the memory block is initialized to zero.

Finally, `free()` is called to deallocate the memory block. The 32-bit memory deallocator may be called by other names in other environments. Simply replace `free()` with a call that is appropriate to your environment.

NULL is always returned from the `FmFree()` function. This is done to help implement the policy that a pointer is either valid or it is NULL (see [§7.12](#)). A pointer variable should never point to an invalid memory object. The memory macros use the NULL return value to store in the memory pointer passed to `FmFree()`.

### **5.2.7 VerifyHeapPointer() Implementation**

#### **VerifyHeapPointer() function**

```
BOOL LOCAL VerifyHeapPointer( LPVOID lpMem )
{
    BOOL bOk=FALSE;
    if (lpMem) {
        WinAssert(FmIsPtrOk(lpMem)) {
```



```

LPPREFIX lpPrefix=(LPPREFIX) lpMem-1;
WinAssert(lpPrefix->lpMem==lpMem) {
    WinAssert(lpPrefix->lpPostfix->lpPrefix==lpPrefix) {
        bOk = TRUE;
    }
}
}
}
return (bOk);
} /* VerifyHeapPointer */

```

VerifyHeapPointer() code first checks to determine whether the pointer passed to it is NULL or not. Next, the lpMem pointer is validated to make sure that it is a valid pointer into the heap. This is done by calling FmIsPtrOk(). Then, a pointer to the prefix structure is obtained and the memory pointer in the structure is checked against the memory pointer being validated. Finally, the prefix pointer in the postfix structure is validated. Most types of memory overwrites past the end of an object trash the prefix pointer in the postfix structure. This is how memory overwrites are detected.

### 5.2.8 FmIsPtrOk() Implementation

The VerifyHeapPointer() must be called only by the heap management functions and is a local function to the heap management module. This function is complete except for the FmIsPtrOk() function. FmIsPtrOk() validates that a pointer passed to it is a realistic heap pointer.

You must code a FmIsPtrOk() function that is appropriate for your environment.

 The FmIsPtrOk() function is defined as follows.

**FmIsPtrOk() function, for Intel segmented protected-mode application**

```

BOOL APIENTRY FmIsPtrOk( LPVOID lpMem )
{
    BOOL bOk=FALSE;

```

```

        _asm xor ax, ax                ;; assume bad selector
        _asm lsl ax, word ptr [lpMem+2] ;; get selector limit
        _asm cmp word ptr [lpMem], ax  ;; is ptr offset under
limit
        _asm jae done                 ;; no, bad pointer
        _asm mov bOk, 1                ;; yes, pointer OK
        _asm done:
        return (bOk);

} /* FmIsPtrOk */

```

The key to this implementation of the `FmIsPtrOk()` function is the usage of `lsl`. This assembly instruction loads a register (in this case, register `ax`) with the limit of the provided selector (in this case, the selector of `lpMem`). If the offset of the `lpMem` pointer is within the limit of the memory segment pointed to by the selector of `lpMem`, the `lpMem` memory pointer is considered valid.

For a non-protected-mode application, the `FmIsPtrOk()` function is as follows.

```

FmIsPtrOk() function, for non-protected-mode application
BOOL APIENTRY FmIsPtrOk( LPVOID lpMem )
{
    return ((lpMem) && (!((long)lpMem & (ALIGNMENT-1))));
} /* FmIsPtrOk */

```

In a non-protected-mode environment, `FmIsPtrOk()` can assume only that the pointer is OK if the pointer is non-NULL and properly aligned.

### 5.2.9 FmWalkHeap() Implementation

An important part of any heap manager is the ability to display a symbolic heap dump.

```

FmWalkHeap() function
void APIENTRY FmWalkHeap( void )

```

```

{
    if (lpHeapHead) {
        LPPREFIX lpCur=lpHeapHead;
        while (VerifyHeapPointer(&lpCur[1])) {
            char buffer[100];
            RenderDesc( lpCur, buffer );
            /*--- print out buffer ---*/
            /* printf( "walk: %s\n", buffer ); */
            lpCur = lpCur->lpNext;
            if (lpCur==lpHeapHead) {
                break;
            }
        }
    }
}

} /* FmWalkHeap */

```

The implementation of FmWalkHeap() is pretty straightforward. It walks the heap getting descriptions of objects and prints them out. The implementation of getting a description of an object is left to RenderDesc().

#### **RenderDesc() function**

```

void LOCAL RenderDesc( LPPREFIX lpPrefix, LPSTR lpBuffer )
{
    if (lpPrefix->lpMem==&lpPrefix[1]) {
        sprintf( lpBuffer, "%08lx ", lpPrefix );
        if (lpPrefix->lpFilename) {
            sprintf( lpBuffer+strlen(lpBuffer), "%12s %4ld ",
                    lpPrefix->lpFilename, lpPrefix->lLineNumber );
        }
        if (lpPrefix->lpClassDesc) {
            sprintf( lpBuffer+strlen(lpBuffer), "%s",
                    lpPrefix->lpClassDesc->lpVarName );
        }
    }
    else {
        strcpy( lpBuffer, "(bad)" );
    }
}

} /* RenderDesc */

```

This is one possible implementation of RenderDesc(). It displays the

address of an object, its filename and line number, if any, and its class descriptor name, if any.

### 5.2.10 FmRealloc() Implementation

Another memory management function that needs a new interface is `realloc()`. While not presented here, the source for `FmRealloc()` can be found in the Code Listings Appendix.

## 5.3 An Interface for Strings

How should strings be dynamically allocated from the heap? Who is responsible for calling the heap interface? The best way to approach this problem is through the utilization of an abstraction layer. It is best to implement the string interface through a set of macros.

### 5.3.1 NEWSTRING() Macro

The first macro is `NEWSTRING()`. It is used to allocate storage for a string from the heap.

```
NEWSTRING() macro
#define NEWSTRING(lpDst, wSize) \
    ( _LPV(lpDst)=FmNew( (SZET) (wSize), NULL, szSRCFILE, __LINE__ ) )
```

`NEWSTRING()` can almost be considered a replacement for `malloc()`. The macro takes two arguments. The first argument is the name of a variable that is to contain the pointer of the allocated block of memory. The second argument is the size in bytes of the block of memory to allocate. This macro was designed this way so that the `_LPV()` macro could be used to avoid type casting of the pointer returned from `FmNew()`.

To free the memory allocated through `NEWSTRING()`, simply call `FREE()`.

### 5.3.2 MYLSTRDUP() Macro

The second macro is MYLSTRDUP(). It is a replacement for strdup() that uses the new heap manager interface.

```
MYLSTRDUP() macro
#define MYLSTRDUP(lpDst,lpSrc) \
    ( _LPV(lpDst)=FmStrDup(lpSrc,szSRCFILE,__LINE__) )
```

MYLSTRDUP() is a macro interface around FmStrDup(). It is a new heap manager function that does the dirty work of duplicating the string.

```
FmStrDup()
LPVOID APIENTRY FmStrDup( LPSTR lpS, LPSTR lpFile, int nLine )
{
    LPVOID lpReturn=NULL;
    if (lpS) {
        SIZET wSize = (SIZET)(strlen(lpS)+1);
        lpReturn = FmNew( wSize, NULL, lpFile, nLine );
        if (lpReturn) {
            memcpy( lpReturn, lpS, wSize );
        }
    }
    return(lpReturn);
} /* FmStrDup */
```

If a NULL pointer is passed into FmStrDup(), a NULL pointer is returned. FmStrDup() works by first calculating the string length and then adding one for the null character. This number of bytes is then allocated by calling FmNew() and finally the string is copied into the new memory buffer.

## 5.4 An Interface for Arrays

Just as with strings, using macros as a level of abstraction greatly simplifies using arrays.

### 5.4.1 NEWARRAY() Macro

The NEWARRAY() macro is used to create a new array with a specific number of array elements.

#### **NEWARRAY macro**

```
#define NEWARRAY(lpArray, wSize) \
    (_LPV(lpArray)=FmNew((SIZET) (sizeof(*(lpArray))*(wSize)), \
    NULL, szSRCFILE, __LINE__))
```

NEWARRAY() takes as its first argument the name of the variable that is to contain the pointer of the allocated array. The second argument is the number of array elements (not bytes) to allocate.

To free the memory allocated through NEWARRAY(), simply call FREE().

### **5.4.2 SIZEARRAY() Macro**

The SIZEARRAY() macro is used to resize an array.

#### **SIZEARRAY Macro**

```
#define SIZEARRAY(lpArray, wSize) \
    (_LPV(lpArray)=FmRealloc((lpArray), \
    (SIZET) (sizeof(*(lpArray))*(wSize)), szSRCFILE, __LINE__))
```

SIZEARRAY() takes as its first argument the name of the variable that points to an allocated array. If it is NULL, SIZEARRAY() allocates a new array. The second argument is the new size of the array in array elements (not bytes).

## **5.5 Detecting Storage Leaks**

If a program allocates an object from the heap, it should also free the object before the program exits. If the object is not freed, it is an error and is called a storage leak.

Do you have any storage leaks in the programs that you write? How do you

detect storage leaks? Unless you have a formal methodology in place, how do you really know?

I once wrote a large program that did not have any formal storage leak detection. The program had no problems and I would have claimed that there were no storage leaks present. However, adding storage leak detection actually turned up a few obscure leaks. They were so obscure that no matter how long the program was run, the storage leaks would never have caused a problem and this is why they were never found.

No matter how small or large a program you write, the program should always have some form of storage leak detection in place.

Storage leak detection should always be in place.
---

The simplest way to detect storage leaks is to call [FmWalkHeap\(\)](#) the moment your program is about to exit. You will then see exactly what objects are left in the heap, if any. Because the filename and line number where the object are allocated is actually displayed, tracking down the storage leak is a snap.

## 5.6 Windows Memory Model Issues

There is a long-standing problem in the Windows environment as to which memory model should be used for developing a program. It is complicated by the fact that most compilers for Windows support four basic memory models. They are small, medium, compact and large. However, the choice is often between the medium memory model and the large memory model because small and compact are too limiting.

### 5.6.1 The Large Memory Model Myth

It is widely believed that the large memory model should not be used for Windows programming. This was true at one time, but today it is a myth!

In Windows, if a program contains more than one data segment, it can be run only once, not multiple times. There is also a limit on how many

memory allocations can be made. This limit is around 4,096 or 8,192, depending upon how Windows was run (standard versus enhanced-mode). This limit is actually a hardware limit in how many segments it has to manage.

These limits existed in Microsoft C6, but beginning with Microsoft C7, these limits were removed. The limits existed due to how the compiler and the run-time libraries of C6 were written.

In C6 with the large memory model, each source file would get its own data segment. However, starting with C7 only one data segment is produced if possible. So this solves the data segment problem.

Concerning the limit on the number of memory allocations, C6 used to eat up a segment on every memory allocation. Starting with C7, the run-time library now employs a subsegment allocation scheme which all but eliminates the problem.

The bottom line is that if you do not mind the overhead of using far pointers extensively (instead of near pointers), go ahead and use the large memory model. It sure makes life a lot easier.

For a more in-depth discussion on the large memory model myth, I suggest you read Windows Internals by Matt Pietrek.

## 5.7 Chapter Summary

- The problem with traditional heap managers is that they assume the programmer never makes a mistake in calling the interface. The traditional heap manager is simply not robust enough for your needs. The solution is to code a module that provides a code wrapper around the existing heap manager.
- Since the new heap manager must also meet the needs of the class methodology and run-time object verification, every block of memory



that is allocated through the new heap manager interface is prefixed and postfixed by a structure.

- NEWSTRING() and MYLSTRDUP() provide a new interface for strings. NEWARRAY() and SIZEARRAY() provide a new interface for arrays.
- Storage leak detection must always be in place in the programs you write. The simplest way to detect storage leaks is to call FmWalkHeap() the moment your program is about to exit.

# Chapter 4: The Class Methodology [Writing Bug-Free C Code](#)

[4.1 The Problem with Traditional Techniques](#)

[4.2 The New Object Model](#)

[4.3 Compile-Time Type Checking](#)

[4.4 Run-Time Type Checking](#)

[4.5 Managing Memory](#)

[4.6 Fault-Tolerant Methods](#)

[4.7 Random Number Generator Source Using Classes](#)

[4.8 A Comparison with C++ Data Hiding](#)

[4.9 Chapter Summary](#)

The class methodology presented in this chapter is the core methodology presented in this book. It solves the data hiding problem. You will produce code that contains fewer bugs by using this methodology because it prevents and detects bugs.

The class methodology helps to prevent bugs by making it easier to write C code. It does this by eliminating data structures (class declarations) from include files, which makes a project easier to understand (because there is not as much global information), which makes it easier to write C code, which helps to eliminate bugs. This class methodology, which uses private class declarations, is different from C++, which uses public class declarations.

The class methodology helps detect bugs by providing for both compile-time and run-time type checking of pointers (handles) to class objects. This run-time type checking catches a lot of bugs for you since invalid object handles (the cause of a lot of bugs) are automatically detected and reported.

All the code introduced in this chapter can also be found in one convenient location in the [Code Listings Appendix](#).

## **4.1 The Problem with Traditional Techniques**

### **4.1.1 Data Structures Declared in Include Files**

The problem in a lot of projects is the number of data declarations in include files. Consider what happens when a small project grows gradually

into a large project. In the small project, you have a small team of programmers who all know the project reasonably well. Each person is contributing code as well as data declarations. Some new data declarations undoubtedly refer to previous data declarations. This methodology works fine for small isolated projects.

What happens as the project grows and more programmers are added to the project? The old programmers continue to code as they always have. The new programmers have a steep learning curve. In order for them to become productive on the project, they first have a lot to learn about how it works. When the new programmer does come up to speed on the project, you now have one more programmer adding information to the pool of information that must now be learned by everyone.

It does not take too long before your pool of information is so large and interconnected that it becomes impossible for any one person to fully understand the project as a whole. At this point, the next logical step is to have individual programmers within the group specialize in a particular area of the code or project. This division of labor can be implemented successfully, but too often it just creates isolated groups with little communication and very little code sharing among groups.

<p>The problem with large development efforts is information overload in the form of data structure declarations that are found in include files.</p>
---

The single biggest problem with any large project is that data structures are declared in include files. Any programming methodology that attempts to solve the information overload issue must also address data structures declared in include files.

#### **4.1.2 Directly Accessing Data Structures**

Once data structure declarations are placed into include files, those declarations become public knowledge.

Do you consider this a good or a bad thing? What are the pros and cons?

The pros and cons of public data structures depend totally upon the size of the project. For small projects isolated to one programmer, public data structures can actually speed up the development time of the project. However, for any moderate to large project, with no matter how many programmers, public data structures quickly becomes a bad thing.

The primary problem with public data structures is that they promote writing code that directly accesses the data structure instead of calling a function that manipulates the data item. This direct access is bad because the distinction between the implementor of the data object and the user of the data object becomes totally blurred.

This blurring between the implementor and user of a data object over time leads to a project that is impossible to modify in any way. Just think what would happen if the data object needed to be changed to support a new feature. All code that directly accesses that data object would have to be changed. This is obviously very undesirable.

Any programming methodology that attempts to solve the information overload issue must also address how data structures are to be accessed.

#### **4.1.3 Compilation Times**

Another problem with public data structures is the time needed to compile source files whenever any change is made to a public structure. Any change to the structure may force you to recompile every source file that references the data structure. Determining this set of files is not always easy and to avoid possible problems you end up compiling the entire program. This actually works quite well for small projects, but what about large projects? What about large projects under version control software? Complete builds of a project may take anywhere from several minutes to several hours.

A change to a single data structure should require, at most, only one source file to be recompiled.

The ideal situation that you should strive for, if at all possible, is a one-to-one dependency between data structure and source file. In other words, a change to a single data structure should require, at most, one source file to be recompiled.

## **4.2 The New Object Model**

Any solution to the information overload problem must address two key issues involving data structure declarations:

*Where are data structures placed?*

*How are data structures accessed?*

### **4.2.1 Terminology**

Before continuing, some terms need to be defined. An object is any data declaration. A method is a function that acts upon an object. A handle is an abstract object identifier. An instance is a unique occurrence of an object that occupies space in memory.

### **4.2.2 Private Objects and Public Methods**

The traditional software development approach is one of public access to objects and public methods. To manipulate the object, you either directly access the object or you call one of the methods of the object.

The new object model I propose is one of private access to objects and public methods. The only way to manipulate an object is by calling one of the methods of the object. This implies that an object has a minimum of two method functions: one method function to create an instance of an object and another method function that destroys an instance.

Private access and public methods also imply that an object's data declaration and method functions are contained in the one source file that implements the object and not an include file.



The object model must support private objects and public methods.

Private objects is just another term for complete data hiding. The data of an object is visible only to the implementor of the object and not to the user of the object.

If this object model can actually be implemented, it will solve the two key issues involving data structure declarations. Where are data structures placed? (in source files, not include files). How are data structures accessed? (privately -- only through method functions).

Let me relate the problem to the real-world problem of generating random numbers. The standard C library routines provide one global random number generator with functions or methods to seed and return the next random number. These functions are `srand()` and `rand()`. How would you extend this model to provide multiple independent random number generators?

One simple extension would be to add another argument to `srand()` and `rand()` that specifies which random number generator to use out of a static array of possible generators. While this design does work, it has several problems. All code that uses the new random number generator interface must cooperate on which indexes to use. What if the random number generator code is in a Windows DLL, which any number of applications can link to and use? Again, all these applications must cooperate on which indexes to use. The static array implementation also imposes a fixed maximum to the number of random number generators that can be used. Therefore, a static array is a bad implementation choice for a generalized object model.

The object model must support an unlimited number of dynamically allocated objects.

What this means is that you need an object model that supports dynamically allocated memory. In terms of the random number generator, you need a data structure containing the data needed to implement the random number that is created and destroyed as needed. The method functions `srand()` and

rand() then operate upon this dynamically allocated data structure.

Creating and destroying objects as needed is also much better than static arrays because all objects now share all available memory. This means that your program is not limited by some arbitrarily picked array bounds but instead by how much memory there is.

### 4.2.3 Windows Object Model

This object model is starting to look like Windows. Think about CreateWindow(), a public method that creates a new window. You have no idea how Windows implements the window, which is a private object, and you are limited in how many windows you can create only by the amount of available (user heap) memory.

However, there are problems with the Windows object model. In SDKs prior to Windows 3.1, all objects had the same data type, namely HANDLE, which was defined to be a WORD. What this means is that if you passed an HPEN to a function that was expecting an HBRUSH, the compiler would certainly not complain. Worse yet, Windows might not even complain at run-time about the type mismatch. What if the function was SelectObject(), which allows any GDI object to be passed to it? Both HPEN and HBRUSH are GDI objects. SelectObject() would be selecting a pen and not a brush. Your program would not perform as expected until you had tracked down the cause of the wrong object being selected.

The object model must support object handles that are type checkable by the compiler.
---

The Windows 3.1 SDK has fixed this problem by allowing you to #define STRICT before including windows.h. What this does is change the type of objects from WORD to a near pointer of a dummy public structure. This in turn allows the compiler to perform type checking.

However, what if you want to implement an object that one of the Windows handles points to? The problem is that the handle is already declared to point to a dummy public structure, but this structure is

obviously not the true implementation. This is ugly and will not work for us, so another technique needs to be found.

### 4.3 Compile-Time Type Checking

We have a requirement for an object model in which there are private objects and public methods that act upon the objects. So how are handles to private objects going to be type checked? Remember, the object is private and the compiler is doing the type checking in modules that do not have access to the object's data declaration.

All objects are dynamic and every object type in the system has at least two method functions. One method creates an object, returning a handle to the object, and another method destroys the object.

A handle could be an index into an object table. It could be a near pointer into a private heap. It could be a long pointer to the object. It could also be a global memory handle in which the object is contained. The point is that the user of a handle does not know and does not need to know exactly what a handle is. To be consistent, however, all objects in an object system usually produce the same type of handle. This prevents some handles from being indexes and some from being memory pointers, which would only confuse the situation.

The user of a handle does not know and does not need to know exactly what a handle is.
--

#### 4.3.1 The Problem

There are almost unlimited numbers of ways to implement this object system except for the requirement that object handles must be type checkable by the compiler. This obviously implies that a handle must be some data type, because only data types can be type checked by the compiler.

If an object's data structure declaration is private and declared and known



only to the one source file that implements the object, how in the world are you going to get the compiler to perform type checking on handles to this object in other source files?

### 4.3.2 The Breakthrough

The breakthrough in accomplishing this complete data hiding while still maintaining compiler type checking came after I realized how to get the C compiler to perform type checking on pointers without knowing what the pointer points to. In other words, it is possible to create a pointer in C that points to an unknown object and yet is type checkable by the compiler. It is also impossible to perform an indirection on this pointer in all source files except in the one source file that implements the object, where an indirection is possible.

Does this sound too good to be true? The remarkable part about this feature of the C language is that you probably have used this feature but never fully realized its potential.

Consider how you would implement a linked list of nodes.

```
Linked list of nodes
typedef struct tagNODE {
    struct tagNODE *pNext;
    ...
} NODE, *PNODE;
```

The solution is to use structure tags. Because PNODE does not even exist when you want to declare pNext, it is declared as a pointer to struct tagNODE. Consider how you would implement two structures that contain pointers to each other.

```
Circular reference problem, first cut
typedef struct tagNODEA {
    struct tagNODEB *pNodeB;
    ...
} NODEA, *PNODEA;
```

```
typedef struct tagNODEB {
    PNODEA pNodeA;
    ....
} NODEB, *PNODEB;
```

Take a close look at this example. In it you see that pNodeB is being declared to be a pointer to a structure tag, a structure that does not yet exist. This example can be rewritten as follows.

**Circular reference problem, second cut**

```
typedef struct tagNODEA *PNODEA;
typedef struct tagNODEB *PNODEB;

typedef struct tagNODEA {
    PNODEB pNodeB;
    ...
} NODEA;

typedef struct tagNODEB {
    PNODEA pNodeA;
    ....
} NODEB;
```

In this example, PNODEA is a type that points to struct tagNODEA and PNODEB is a type that points to struct tagNODEB. PNODEA and PNODEB are then used even though the structure declarations do not exist yet.

Structure tags allow you to create a pointer to an object before the object even exists and perform type checking on the pointers. In fact, the pointer declarations could have been placed in an include file and used by other source files. As long as the other source files do not try to perform an indirection on the pointer, everything works. Then in the source file with the structure declaration (with the appropriate structure tag) pointer indirections have meaning. Pointer indirections have meaning when they appear in a source file that has a structure declaration with the appropriate structure tag.

---

Structure tags allow you to create a pointer to an object before the object even exists and perform type checking on the pointers.

### 4.3.3 NEWHANDLE() Macro

Now that we know this, we can write a macro that introduces new type checkable handles into the programming environment.

**The NEWHANDLE() macro**

```
#define NEWHANDLE(Handle) typedef struct tag##Handle *Handle
```

The NEWHANDLE() declarations are almost always placed in an include file that gets included into all source files. NEWHANDLE() is usually not used in source files.

Notice in NEWHANDLE() how the token pasting operator (##) is being used in tag##Handle to create a structure tag that is derived from the handle name. By convention, all handle types must be in uppercase and prefixed with the capital letter H (HRAND, for example).

If your C environment does not support the token pasting operator, but your preprocessor follows the Reiser model, you can still accomplish token pasting. See [§2.2.8](#) for details. This technique involves replacing ## with /\*\*/.

Going back to the random number generator example, creating a handle called HRAND would now be easy.

**HRAND handle declaration**

```
NEWHANDLE (HRAND) ;
```

**NEWHANDLE(HRAND) macro expansion**

```
typedef struct tagHRAND *HRAND;
```

So, HRAND is really just a pointer to an unknown structure whose structure tag is tagHRAND.

The HRAND type can be used even though no structure with a structure tag of tagHRAND exists in the modules being compiled. This is just like the linked list of nodes with the PNODEA and PNODEB types.

A complete random number generator interface specification in an include file could quite possibly be written as follows.

```
Random number generator interface
NEWHANDLE (HRAND) ;
.
.
.
EXTERNC HRAND APIENTRY RandCreate ( int );
EXTERNC HRAND APIENTRY RandDestroy ( HRAND );
EXTERNC int APIENTRY RandNext ( HRAND );
```

It is important to realize how the HRAND data type works. Immediately after the NEWHANDLE(HRAND) declaration, HRAND can be used just like any other data type except that it cannot be dereferenced because what HRAND really is is not yet known. In other words, HRAND can be used in function prototypes and HRAND variables can be initialized and passed around, but trying to dereference the HRAND variable will not be possible.

Consider some code that needs its own random number generator. It creates one using RandCreate(), uses it by calling RandNext() and when finished, calls RandDestroy(). The code is able to use HRAND without knowing what HRAND points to. There can also be an unlimited number of random number generators active at any given time.

```
Function that uses a random number generator object
void Testing( void )
{
    HRAND hRand=RandCreate(0);
```

```

    LOOP(100) {
        printf( "Number %d is %d\n", loop, RandNext(hRand) );
    } ENDLLOOP
    hRand = RandDestroy( hRand );

} /* Testing */

```

It is important to realize how the HRAND data type is working in Testing(). The Testing() function is using an HRAND variable hRand even though Testing() has no idea what hRand points to or how the HRAND data type is implemented. In fact, the HRAND structure declaration is not even visible to this Testing() function. This is because HRAND at this point is a pointer to an unknown, but named (tagHRAND), object/structure.

Notice the spelling of hRand. It is an upper- and lowercase variant of its data type, HRAND. You should always try to derive object variable names from object data types this way.

The only source line that may not be totally clear in Testing() is the hRand=RandDestroy(hRand); line. By convention, all functions that destroy an object return the NULL object, so this ends up setting hRand to NULL.

All object destroy functions return the NULL object.

The reasoning behind this is that you always want a handle variable to contain a valid handle or NULL. You never want a handle variable to be uninitialized or contain an old, previously valid, handle (see [§7.12](#) for more information on the usage of NULL).

#### 4.3.4 Implementing the Random Number Generator

There are still a lot of loose ends to fully implement the random number generator, but here is a rough shell of what the code will look like.

```

Random number generator implementation, first cut
typedef struct tagHRAND {
    long lRand;

```

```

};

HRAND RandCreate( int nSeed )
{
    HRAND hRand;
    (allocate memory);
    hRand->lRand = nSeed;
    return (hRand);
} /* RandCreate */

HRAND RandDestroy( HRAND hRand )
{
    (free hRand memory)
    return (NULL);
} /* RandDestroy */

int RandNext( HRAND hRand )
{
    hRand->lRand = NEXTRAND(hRand->lRand);
    return (FINALRAND(hRand->lRand));
} /* RandNext */

```

The struct tag `HRAND` structure declaration is declared in the source file that implements `HRAND` and not in an include file.

The implementation is straightforward. The details of random number generation in the `NEXTRAND()` and `FINALRAND()` macros and how memory is allocated and freed for the objects has been left out. This is discussed later.

The only catch is `TYPEDEF`. For standard C, `TYPEDEF` is defined to be typedef. For C++, `TYPEDEF` is defined to be nothing. This was done to avoid the Microsoft C8 warning message C4091 no symbols were declared under C++.

In the struct tag `HRAND` declaration, a declarator is not required after the ending brace and before the semicolon because a structure tag is being used. When a structure tag is used, the declarator is optional. Without a structure

tag, the declarator is generally required. This has to do with how C works and it is spelled out in §A8 of [The C Programming Language](#).

This random number generator source is contained in its own source file. It is important to realize this. This code need not and should not be declared along with other code, like the Testing() function, that uses the random number generator. The implementation of an object should be contained in a separate source file and a source file should implement, at most, a single class object.

The implementation of an object is contained in its own source file and is separate from code that uses the object.
---

This implementation is in its own source file and #includes the same include file that all other source files include. This include file contains the NEWHANDLE(HRAND) declaration and function prototypes.

Even in the source file that implements the HRAND object, the compiler has no idea what HRAND is until a structure is declared with a structure tag of tagHRAND. At this point, the compiler binds what HRAND is to the tagHRAND structure and indirections are now possible. In other words, as soon as this binding of HRAND to the structure with tag tagHRAND takes place, we are free to implement the HRAND object because indirections on the object are now possible.

Indirections on a handle pointer are valid only in the module that implements the class object.
---

#### 4.3.5 Summary

The problem in an object model with private objects is getting the compiler to type check pointers to the objects when the objects are not even known to the compiler. The solution is to use an incomplete type, a feature of C. A handle to an object is a pointer to a structure that has a structure tag, but a structure that does not have a body.

This incomplete type allows the compiler to perform type checking on a

pointer to the structure when the structure is not known. The `NEWHANDLE()` macro introduces a new type checkable handle into the system.

## 4.4 Run-Time Type Checking

An important part of any object system is to provide as much error detection and reporting as possible. A common mistake that all C programmers make is accidentally passing an incorrect value to a function. In the case of handles, which are simply memory pointers, passing an incorrect value to a method function may or may not have unpredictable results because the memory pointer may just happen to be valid (but pointing to the wrong memory location).

Ideally, passing an incorrect pointer to a method function causes some sort of protection fault which would allow you to track down the problem. What if using an incorrect pointer does not cause a fault? The method function more than likely ends up trashing memory instead; in any case, it will not perform the function the caller intended.

A prime example of how this can happen is using an object handle after the object has been destroyed. The memory pointed to is more than likely still addressable, but there is no valid object in the memory. Another example is memory that has been accidentally overwritten. If the memory contains any object handles, those object handles are now invalid. By far the most common memory overwrite is writing beyond the end of a character array. Detecting this is discussed in [Chapter 5](#).

The first line of defense against incorrect handles is to have the compiler perform type checking on the handles. This makes it almost impossible at compile-time, except through type casting, to pass an incorrect handle to a method function.

The second line of defense is to have every function that accesses an object verify that the object is an object of the correct type at run-time. In our new object model, the functions that access the object are the method functions, all of which are contained in one source file. Because all the method



functions are localized to one file, this opens up interesting optimization possibilities in performing run-time type checking.

An object system that performs run-time type checking on all objects passed to method functions catches a lot of programming mistakes automatically.

#### 4.4.1 Requirements

Adding run-time type checking into a system is not a new idea nor is it a hard thing to do. However, adding it into a system almost transparently is a challenge.

*Low overhead.* The first requirement is that run-time type checking must have a minimal impact on the execution time of a program. A 1 percent or less impact would be great. A 10 percent impact would be substantial, but it could be justified.

*Fault-tolerant code execution.* Any syntax we come up with must be able to support conditional execution of a section of code. If the run-time object verification succeeds, you want the code to execute. If the verification fails, you do not want the code to execute. What good would it be to have a system that notifies you of an object verification failure only to bomb a split second later on code that expected a valid handle but did not have one?

*Automatic and easy to use.* Any system that we come up with must not require a lot of work on the part of the programmer. The goal is to make the programmer's job easier, not harder.

*Minimal code and syntax changes.* Again, we do not want to create a system that is hard for the programmer to use. Any system we come up with must not require a lot of code changes.

*Does not change the sizeof() an object.* A system that changes the sizeof() an object simply because it is being type checked is undesirable and should be avoided.

*Must itself not cause crashes.* This may seem obvious, but a system must be able to withstand any address passed to it for verification, even addresses that are invalid. It is not enough to simply verify that an object at a valid address is the correct object. It must also make sure that the address itself is valid before attempting to validate the object. This is hard to implement since it requires explicit knowledge of the memory architecture of the hardware you are running on.

*Withstands implementation changes.* We do not want the run-time type checking to be hard-coded. Instead, it should be isolated through the use of macros. This allows the run-time type-checking implementation to radically change with no source code changes in the modules that use run-time type checking.

#### **4.4.2 What to Use for Data Type Identification**

There appears to be a contradiction in the requirements section. How can type checking be added to an object and not have the object change size? The solution is to let the layer beneath the objects keep track of object types. This layer is the heap manager and, as we found out earlier, the standard heap management routines are not robust enough and would have to be replaced anyway. Why not, then, just add one more argument to the memory allocation routine that indicates the type of the object being allocated? This covers all instances of objects in the class methodology, since all objects are dynamically allocated in the heap.

The big question now is what to use to indicate the type of an object. We could use a unique integer value, but this is not automatic. It requires the programmer to maintain the list of IDs. Worse yet, in a shared DLL situation, the programmer likely has multiple applications using the same DLL, so what unique integer values are used in this case? Now all applications need to cooperate on the IDs to use. Since this is undesirable, unique integer values will not work.

**Using type information, a bad implementation**

```
static int nTypeOfHRAND=(hard-coded number);  
.
```

```

.
.
HRAND RandCreate( int nSeed )
{
    HRAND hRand=(allocate memory using nTypeOfHRAND);
    hRand->lRand = nSeed;
    return (hRand);
}

```

The goal is to have the object system automatically determine the type identifier of an object. One possible solution would be to have the heap manager generate unique type IDs at run-time as needed. While this would certainly work, there is a much better way. Remember that the entire goal is to come up with any guaranteed unique number as the type identifier.

Why not just use the address of nTypeOfHRAND as the type identifier! The address is absolutely unique. In fact, you could use the address of anything that is uniquely associated with the class object.

So, we will create a class descriptor structure that contains information about a class object and use its address as the type identifier. There will be one class descriptor per class.

```

A class descriptor
typedef struct {
    LPSTR lpVarName;
} CLASSDESC, FAR*LPCLASSDESC;

```

Associating the variable name typically used for instances of a class is a valuable piece of information to associate with the class description. This is done with the lpVarName member of the CLASSDESC structure.

This allows the custom heap manager to produce symbolic dumps of the heap, complete with variable names used in the code.

An object's type identifier is simply a pointer to its class descriptor structure.

How are these class descriptors going to be named?

#### 4.4.3 Naming Class Descriptors

The class descriptor address is needed during run-time object verification and it is needed when the object is created. Is there a way that its address can be obtained automatically, without having to specify the actual address by explicit reference?

Consider the random number generator example. The data type is HRAND and instance variables are named hRand. We want to run-time type check the variable name hRand, not the data type HRAND. Provided the class descriptor name contains hRand, the address of its class descriptor can be determined automatically! How? Through the use of the C preprocessor token pasting operator.

We now need a macro that, when given a variable name, provides us with the name of its class descriptor. The `_CD()` macro does this for us.

```
The _CD() macro, for use only in other macros  
#define _CD(hObj) hObj##_ClassDesc
```

Notice in `_CD()` how the token pasting operator (`##`) is being used in `hObj##_ClassDesc` to derive the class descriptor name from the object name.

The `_CD()` macro begins with a underscore character. This indicates that the macro is to be used only in other macros, not in source code. See [§3.4.1](#) for more information on naming macros.

The key to providing object-based macros is realizing that the name of an object's class descriptor must be based upon the variable name used in the code and not on the data type of the object.

Basing a class descriptor name on an object's variable name instead of the object's data type is a powerful concept. It allows us to write macros that are

object-based. The only piece of information needed is the actual variable name. All other information can be obtained from the class descriptor.

```
Using _CD() for the hRand object
_CD(hRand)

_CD(hRand) macro expansion
hRand_ClassDesc
```

In the case of the hRand object, the class descriptor for hRand is named hRand\_ClassDesc.

We are now ready to allocate and initialize the class descriptor.

#### 4.4.4 The CLASS() Macro

The class descriptor structure used for run-time type checking needs to be allocated and initialized once. There is one class descriptor per class. It makes a lot of sense to do this at the same place in the code where the class structure is being declared. The class descriptor for the random number generator object would look like the following.

```
HRAND class descriptor
static CLASSDESC _CD(hRand)={"hRand"};
```

We can now design a macro that performs all of class descriptor and structure declaration dirty work in one step.

```
The CLASS() macro
#define CLASS(hObj,Handle) \
    static CLASSDESC _CD(hObj)={#hObj}; TYPEDEF struct
tag##Handle

HRAND using CLASS() macro
```

```
CLASS(hRand, HRAND) {  
    long lRand;  
};
```

The CLASS() macro is used only by source files that implement an object. The CLASS() macro is never used in include files.

The CLASS() macro takes two arguments. The first argument is the variable name that is used to represent instances of objects of this class. The second argument is the handle name of the class objects. The class descriptor is allocated and initialized based upon the variable name and the stringizing operator (i.e., #hObj). The structure declaration is started based upon the handle name (i.e., TYPEDEF struct tag##Handle).

Allocating memory for a class object based upon its variable name now becomes incredibly simple. In the case of an hRand variable name, the number of bytes that need to be allocated is sizeof(\*hRand) and the type information address is &\_CD(hRand).

We are now ready to implement run-time type checking.

#### 4.4.5 The VERIFY() Macro

Given any valid variable name that is a handle to an object, an ideal syntax for the run-time object verification macro would be as follows.

##### **Ideal VERIFY() macro syntax**

```
VERIFY(hObject);
```

*or*

```
VERIFY(hObject) {  
    (block of code)  
}
```

##### **VERIFY() and VERIFYZ() macros**

```
#define VERIFY(hObj) WinAssert(_VERIFY(hObj))
```

```
#define VERIFYZ(hObj) if (!(hObj)) {} else VERIFY(hObj)
```

The VERIFY() macro is designed to be used only by the source file that implements an object, not by other source files that just use an object.

At the core of the VERIFY() macro is its usage of [WinAssert\(\)\\_§3.3](#). This allows VERIFY() to be terminated with either a semicolon or a block of code.

Again, notice that the only piece of information needed is the object's variable name. No other information needs to be provided. The VERIFY() macro implements the syntax that is desired but leaves the implementation to another macro called \_VERIFY().

The VERIFYZ() macro is a slight variation on the VERIFY() macro. If a NULL pointer is passed to VERIFYZ(), the optional body of code is not executed, nor is this treated as an error. VERIFYZ() is useful in allowing NULL pointers to be passed to an object's destroy method.

Given a handle to an object, which is just a pointer to the object, you should be able to obtain information about the object maintained by the heap manager. As we will see in [Chapter 5](#), the heap manager just provides a wrapper around the object. This means that the heap manager's information about the object can be accessed by using negative offsets from the object pointer. For speed, these offsets are known by both the heap manager code and the run-time object verification code. (See Figure 4-1).

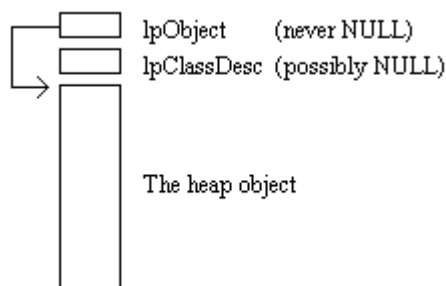


Figure 4-1: Memory layout of a heap object.

The data item immediately before a valid heap object is a long pointer to the class descriptor of the object or NULL, which indicates that no class

descriptor exists. The data item before the class descriptor pointer is a pointer to the heap object, which is used for heap pointer validation.

Using hRand as an example, the steps needed to verify that the address contained in hRand does indeed point to a valid random number object are as follows.

1. Is hRand a valid pointer into the heap? This step is the most difficult since it depends upon the machine architecture that the program is running on. More on this later, but for now we will use FmIsPtrOk(hRand).
2. Does the address in hRand match the address at hRand minus two data items? Namely, (((LPVOID)hRand)==\*(LPVOID FAR\*) ((LPSTR) hRand-sizeof(LPCLASSDESC)-sizeof(LPVOID))).
3. Does the address at hRand minus one data item match &\_CD(hRand)? Namely, ((&\_CD(hRand))==\*(LPCLASSDESC FAR\*)((LPSTR)hRand-sizeof(LPCLASSDESC))).

One possible \_VERIFY() macro implementation is as follows.

```
_VERIFY() macro
#define _S4 (sizeof(LPCLASSDESC))
#define _S8 (sizeof(LPCLASSDESC)+sizeof(LPVOID))
#define _VERIFY(hObj) \
    ( FmIsPtrOk(hObj) && \
      ((LPVOID)hObj)==*(LPVOID FAR*) ((LPSTR)hObj-_S8)) \
      && ((&_CD(hObj))==*(LPCLASSDESC FAR*) ((LPSTR)hObj-_S4)) )
```

To be efficient, the \_VERIFY() implementation must be tailored to a specific development environment. It also assumes that an effective FmIsPtrOk() can be written. This will be discussed in [Chapter 5](#). I have found out over the years that the source code has stayed the same, but the \_VERIFY() macro implementation keeps on changing to suit my development environment.

To be efficient, the \_VERIFY() implementation must be tailored to a



specific development environment.

My development environment was once based upon the small memory model of the Microsoft compiler. Then it moved to the medium memory model; then to a based heap allocation scheme and finally to a model in which data is kept in far data segments. Through each of these changes, the code has stayed the same, but the `_VERIFY()` implementation has changed quite a bit.

You must code a `_VERIFY()` that works in your particular environment.

I cannot provide you with a generalized `_VERIFY()` implementation. You must code a `_VERIFY()` that works in your particular environment. The `_VERIFY()` that I use in my environment follows.

#### 4.4.6 My `_VERIFY()` Macro

My `_VERIFY()` macro is tailored to the segmented architecture of the Intel CPU and is highly optimized. It assumes that a program was developed using the medium memory model and that object handles are 32-bit segment/offset pointers.

```
My _VERIFY() macro  
#define _VERIFY(hObj) Verify_##hObj((long)hObj,  
(WORD) &_CD(hObj))
```

My implementation of `_VERIFY()` ends up calling a local (near) function whose arguments are passed using the register calling convention. I turned the code into a function call, because I was dissatisfied with the speed (too slow) and size (too big) of the code generated by the compiler for the macro form of `_VERIFY()`. I discovered this by using the code generation option (`/Fc`) of the Microsoft C8 compiler. The function call saves code size and since the call is a near call using the register calling convention, the speed is actually quite good. The `CLASS()` macro was changed slightly to automatically prototype the `Verify_##hObj` function for me.

The 32-bit object pointer is type cast into a long because 32-bit pointers cannot be passed through the register calling convention, but a long can. The 32-bit class descriptor address is type cast to a WORD for two reasons. First, because the register calling convention does not allow for two long values to be passed through registers, but it does allow a long and a WORD. Second, because the medium memory model is being used, the segment for all class descriptors has the same value, so it is ignored and only the lower 16 bits (the offset) are used for type checking.

**The verification code used by my `_VERIFY()` macro**

```
; DX:AX = far pointer to verify
; BX     = offset to object class descriptor
;
; WARNING: This code assumes the register calling convention
; used by Microsoft C8. It may change in future compiler
; versions.
    xchg ax, bx
    xor cx, cx                ;; assume a bad selector
    lsl cx, dx                ;; verify selector, length
    cmp bx, cx
    mov cx, 0                 ;; assume false return
    jae done                  ;; long pointer was bad
    mov es, dx
    cmp word ptr es:[bx-8], bx ;; test offset
    jne done
    cmp word ptr es:[bx-6], dx ;; test segment
    jne done
    cmp word ptr es:[bx-4], ax ;; test class desc offset
    jne done
    inc cx                    ;; true return
done:
    mov ax, cx
    ret
```

This verification code is really part of a macro that is used by an assembly file that creates the properly named code segment and verification code so that it can be called as a near function. This assembly file is part of my project makefile. It uses an inlining file feature of the makefile to accomplish this.

The execution overhead of the verify code that I use is low because it has been handwritten in assembly. A fair estimate is that one verify takes 66 clock cycles. Assuming that you run the code on an Intel 66-MHz 80486, you can perform one million object verifications per second. A 1 percent processor overhead would require 10,000 object verifications per second. The application that I wrote usually does less than 10,000 object verifications per second (as measured by changing the `_VERIFY()` macro to increment a counter), so I know that the overhead is less than 1 percent of the processor.

This implementation of `_VERIFY()` takes full advantage of the features of my own environment to meet my demanding speed and space requirements.

#### **4.4.7 Summary**

The `CLASS()` and `VERIFY()` macros work together to provide what is needed to run-time type check object handles. The stringizing operator and the token pasting operator are key features of C that make these macros so easy to use.

### **4.5 Managing Memory**

What should be the interface for allocating and freeing objects? The interface should probably be implemented through a set of macros to allow the implementation to change without having to change any source code.

#### **4.5.1 NEWOBJ() and FREE() Interfaces**

A model for allocating an object is `NEWOBJ(hObj)`. The `NEWOBJ()` macro implementation should do all the dirty work of allocating the memory from the heap manager, passing the appropriate type information and assigning the memory pointer to `hObj`.

A model for freeing an object is `FREE(hObj)`. It should call the heap manager to free the memory associated with `hObj`. It should also ensure that `hObj` is set to `NULL` as well. This allows us to find any bugs that involve

using the handle after calling `FREE()`, because dereferencing a far `NULL` pointer causes a CPU fault to occur in protected-mode architectures. See [§7.12](#) for more information on using `NULL`.

However, before we can write these macros, the interface to the new heap manager must be specified.

### 4.5.2 Heap Manager Interface Specification

Because the heap manager is at the core of the object management system, it should have as much error checking information available in it as possible. One piece of information we already know it must have is the address of a class descriptor. Since this allows us to write a heap manager that provides great symbolic dumps of the heap, why not add some more information that would be meaningful in the heap dump?

Why not include the filename and line number where the object was allocated? This information is useful for non-object heap objects like strings. The reason that it is not as useful for objects is that objects are created only in one method function.

Another concern is which memory model to use for heap objects. For specialized applications, this is of major concern since the memory model affects the performance of the application. However, for the object model, an interface that uses 32-bit pointers is assumed. The 32-bit address may be a segment and offset for segmented architectures, or it may be a linear virtual address in flat-model architectures. Whichever architecture it is, it does not matter.

#### **The heap manager interface**

```
EXTERNC LPVOID APIENTRY FmNew ( SIZET, LPCLASSDESC, LPSTR,  
int );  
EXTERNC LPVOID APIENTRY FmFree ( LPVOID );
```

The `FmNew` (far memory new) takes four arguments. The first argument indicates the number of bytes to allocate in the object. It is of type `SIZET`.

Under most C environments, this will be defined to be `size_t`. The second argument is a pointer to a class descriptor or `NULL` if no class descriptor exists. The third and fourth arguments specify the filename and line number where the `FmNew` call took place. The return value is a long void pointer to the allocated memory.

The `FmFree` (far memory free) takes one argument. The argument is a memory object that was previously allocated through `FmNew()`, or `NULL`. The return value is a long void pointer that is always `NULL`.

The heap manager is discussed in further detail in [Chapter 5](#). For now, this gives us enough information to implement the `NEWOBJ()` and `FREE()` macros.

### 4.5.3 NEWOBJ() and FREE() Implementations

Now that the heap manager interface has been specified, the `NEWOBJ()` and `FREE()` macros can be designed.

```
NEWOBJ() and FREE() implementation, first cut
#define NEWOBJ(hObj) \
    hObj = FmNew(sizeof(*hObj), &_CD(hObj), __FILE__, __LINE__)
#define FREE(hObj) hObj = FmFree(hObj)
```

Notice that `hObj` is the only piece of information needed by `NEWOBJ()` and `FREE()`. The size, in bytes, of the object pointed to by `hObj` is `sizeof(*hObj)`. The address of the class descriptor for `hObj` is `&_CD(hObj)`. Finally, the filename and line number of the memory allocation are simply `__FILE__` and `__LINE__`. This implementation does in fact work quite well except for two minor problems.

The first problem is with `__FILE__`. Every time it is used, it introduces a new string into the program. However, a solution exists. Use the filename variable that is used by the [WinAssert\(\) §3.3](#) code. The variable is named `szSRCFILE`. You just have to make sure that `USEWINASSERT` is placed at the top of the source file.

Every source file should have a USEWINASSERT at the top of the source file.

The second problem is with the differences between C and C++. The first pass implementation works just fine in C but not in C++. It involves the usage of void pointers. In C, a void pointer may be legally assigned to a typed pointer. In C++, this is illegal without the appropriate type cast. We do not want to have to pass in the data type of the object, since this would ruin the slick implementation of NEWOBJ() and FREE().

Instead of type casting the right-hand side to the correct data type, why not try to type cast the left-hand side to a void pointer type? How can this be done?

```
_LPV() macro  
#define _LPV(hObj) *(LPVOID FAR*)&hObj
```

This \_LPV() macro effectively changes the type of an l-value object to LPVOID. The danger in this macro is that it assumes the argument is a far pointer to an object.

We can now rewrite the NEWOBJ() and FREE() macros as follows.

```
NEWOBJ() and FREE() implementation, final form  
#define NEWOBJ(hObj) \  
  
    (_LPV(hObj)=FmNew(sizeof(*hObj),&_CD(hObj),szSRCFILE,__LINE__))  
#define FREE(hObj)  (_LPV(hObj)=FmFree(hObj))
```

Regardless of the type of hObj, it is forced into an LPVOID type so that an assignment can be made to hObj without compiler error or warning messages.

#### 4.5.4 Summary

The NEWOBJ() and FREE() macros work together to provide an abstraction layer on top of the heap manager code that allows objects to be created and destroyed.

## 4.6 Fault-Tolerant Methods

An important part of the object model presented in this chapter is that it allows us to write code that checks the validity of object handles passed to method functions at run-time. The [VERIFY\(\) syntax](#) allows for a block of code to be conditionally executed depending upon the validity of an object handle. This allows a certain degree of fault-tolerance to be built into code.

Protect the code of a method function in the body of a VERIFY() block.

If you are careful in how you design method functions, your program is able to withstand any number of faults, indicating programming errors or bugs, but your program remains running.

Without giving consideration to the fact that a method function may fail, a program may end up bombing anyway. So, how should method functions be designed to withstand faults?

A model that I use for designing method functions is to treat objects like state machines and methods as state machine transitions.

### 4.6.1 The State Machine Model

A state machine consists of a number of valid states and ways to move from one valid state to another valid state.

Objects are state machines and methods are state machine transitions.

The important part to remember about this model is that an object instance is always in a valid state. What happens when a method function fails?

Consider an object that is in a valid state. We wish to execute a method

function on the object. If the handle passed to the method function is valid, the method function executes and takes the object from one valid state to another valid state. If the handle passed to the method function is invalid, the method function does not execute and all objects in the heap stay in their current valid state.

What this means is that method functions must never leave the object in an invalid state. The subtle implication of this is that an object must never require more than one method function to be called to take the object from one valid state to another, because if the first method function call succeeds, but the second method function call fails, the object is left in an invalid state.

When a method function fails due to an invalid handle, all objects in the heap stay in their current valid state.

#### 4.6.2 Designing Method Functions

Designing method functions to be fault-tolerant when the method function returns no information is a snap.

```
Fault-tolerant method function, no return information
void APIENTRY Method( HOBJECT hObject, (other arguments) )
{
    VERIFY(hObject) {
        (body of code)
    }
} /* Method */
```

Because the method function returns no information, making it fault-tolerant simply means enclosing the body of the method in a VERIFY() block.

How do you design a method function to fail gracefully when the method function returns information? At first, this may seem impossible, but in practice I have found it to be an easy task.



If the method function fails, setting the return information to a value that is reasonable is OK. If the method function is returning a simple numeric value and zero is a possible value, return zero. If a character buffer is being returned, return a null string or whatever string would be considered valid.

**Fault-tolerant method function, with return information**

```
TYPE APIENTRY Method( HOBJECT hObject, (other arguments) )
{
    TYPE var=(failure value)
    VERIFY(hObject) {
        (body of code)
        var = (success value)
    }
    return (var)
} /* Method */
```

The goal in a failure case is to return information that is reasonable. This way the code calling this method function never knows that the method function failed due to a bad memory pointer, which more than likely would have crashed your program anyway.

Consider how to create a fault-tolerant RandNext().

**Fault-tolerant RandNext() method function**

```
int APIENTRY RandNext( HRAND hRand )
{
    int nRand=0;
    VERIFY(hRand) {
        hRand->lRand = NEXTRAND(hRand->lRand);
        nRand = (int)FINALRAND(hRand->lRand);
    }
    return(nRand);
} /* RandNext */
```

In the case of the RandNext() method function, the failure case is to return zero. While zero is admittedly not random, at least the program using the

random number generator is not going to bomb and you will be notified of the run-time object verification failure.

### 4.6.3 Summary

In practice, I have found writing method functions following the state machine model to be a highly effective means of writing a fault-tolerant program.

You may be wondering if writing method functions that are fault-tolerant is even worth it. After all, if a method function fails, that means that an object handle is invalid. And if an object handle is invalid, won't the invalid handle just cause an onslaught of failures?

During development, yes, an onslaught of failures generally occurs, but what about when the program is in its final shipping form? It has been my experience that most faults in a released program cause only a few failures.

Most faults in a shipping product do not cause an onslaught of failures.
--

Isolating and recovering from these failures using run-time object verification allows the program to continue running.

## 4.7 Random Number Generator Source Using Classes

It is time to bring together everything that has been learned in this chapter and rewrite the random number generator.

The interface specification (NEWHANDLE() and function prototypes) remains the same and is contained earlier in this chapter. The final implementation of the random number generator source that uses the macros defined in this chapter is as follows.

```
Random number generator implementation, final version
#include "app.h"
```

```

USEWINASSERT

CLASS(hRand, HRAND) {
    long lRand;
};

HRAND APIENTRY RandCreate( int nSeed )
{
    HRAND hRand;
    NEWOBJ(hRand);
    hRand->lRand = nSeed;
    return (hRand);
} /* RandCreate */

HRAND APIENTRY RandDestroy( HRAND hRand )
{
    VERIFYZ(hRand) {
        FREE(hRand);
    }
    return (NULL);
} /* RandDestroy */

int APIENTRY RandNext( HRAND hRand )
{
    int nRand=0;
    VERIFY(hRand) {
        hRand->lRand = NEXTRAND(hRand->lRand);
        nRand = (int)FINALRAND(hRand->lRand);
    }
    return(nRand);
} /* RandNext */

```

The CLASS() macro is used in the source file that implements the HRAND class object, not in an include file.

This random number generator implementation is contained in its own source file or module separate from all other modules. This ensures that the HRAND implementation is known only to the functions that implement the random number generator and is not known to functions that simply use random numbers.

The interface specification for this random number module is contained in `app.h` and is accessed through `#include "app.h"`. The interface specification contains the `NEWHANDLE(HRAND)` declaration and prototypes for `RandCreate()`, `RandDestroy()` and `RandNext()`.

`USEWINASSERT` allows the code to use the `WinAssert()` macro, which is used by the `VERIFY()` and `VERIFYZ()` macros, and makes the current filename known through the `szSRCFILE` variable, which is used by the `NEWOBJ()` and `WinAssert()` macros.


The `CLASS()` macro allocates and initializes a class descriptor for `HRAND` and binds the `HRAND` handle to an actual data structure. The class descriptor is used by the `NEWOBJ()`, `VERIFY()` and `VERIFYZ()` macros. The binding of the `HRAND` handle to an actual data structure allows us to implement the random number generator, because indirections on `hRand` are now possible.

`RandCreate()` uses `NEWOBJ()` to create a new object, initializes it and returns the handle to the caller.

`RandDestroy()` performs run-time object verification on the `hRand` variable by using `VERIFYZ()`. If `hRand` is non-zero and valid, the object is freed by using `FREE()`. Finally, `NULL` is returned because all destroy methods return `NULL` by convention.

`RandNext()` performs run-time object verification on the `hRand` variable by using `VERIFY()`. If `hRand` is valid, a new random number is generated. Finally, the next random number (or an error random number of zero) is returned.

There is a lot going on behind the scenes in this code. The object-oriented macros are actually hiding a lot of code. It is instructive to see everything that is going on by running this source through the preprocessor pass of the compiler and viewing the resulting output.

 In Microsoft C8, this is done with the `/P` command line option and results in an `.i` file.

## 4.8 A Comparison with C++ Data Hiding

C++ does have a lot to offer (inheritance and virtual functions), but one of the things I do not like about C++ is that it does not allow for the complete data hiding of class declarations. For example, in order to use a class, you must have access to the full declaration of the class. To change the implementation (private part) of a class means that more than likely you have to recompile all source files that just use the object.

This gets even more complicated when inheritance is used and a class implementation (private part) is changed because all classes that are derived from the changed class have definitely changed. This will cause a recompile of a lot of code when all you did was change the private part of one class. The bottom line in C++ is that the private parts of classes are not so private! I do not consider the private part of a C++ class to be complete data hiding.

The data hiding problem is one of the reasons that the class methodology was developed. The problem with almost every large project is that there is simply too much information in the form of data (class) declarations. This results in a project that is hard to work on because of the information overload. The class methodology allows every data (class) structure to be completely hidden. This is done by moving data declarations out of include files and into the modules that implement the objects.

Take, for example, the random number generator just discussed. Users of a random number generator can see only the random number generator interface and nothing more. They see the HRAND data type and the prototypes of the method functions but not the implementation. In fact, the implementation can change totally and only the one source file that implements the random number generator needs to be recompiled. This is because the implementation (class declaration) is declared in only the one source file that implements the object. This is a powerful concept when applied to an entire project.

### 4.8.1 Another View

In §13.2.2 (Abstract Types) of [The Design and Evolution of C++](#), Stroustrup laments that the data hiding view I have expressed above is a common view about C++ but that it is wrong. I disagree. Stroustrup goes on to explain how data hiding can be accomplished in C++ by using abstract classes.

The solution involves using two classes: a base class and a derived class. The base class is declared in an include file to be an abstract class. This declares the interface to the object (not the data) which is visible to all users of the object. The derived class is declared in the source file that implements the object. It includes the (private) data and is derived from the abstract base class. This derived class is the real object, which is invisible to all users of the object.

However, Stroustrup fails to point out the problems in using abstract types to perform data hiding in C++.

*Problem one:* Creating a new instance. Code that creates an instance of the class must have access to the derived class declaration. Therefore, code that uses the class cannot use the new operator to create a new instance of the class because the code has access to only the abstract base class, not the derived class. One possible solution is to declare a static member function in the abstract base class that is implemented in the derived class module. This function can then create a new derived object, returning a pointer to the base class.

*Problem two:* All method functions must be virtual. All functions that interface to the object must be declared as virtual, which adds function call overhead. Therefore, calling a function declared in the base class is really calling a derived class function. Consider what would happen if the functions were not virtual. They could be implemented, but how would the base class functions access data in the derived class? The problem is that the this pointer in the base class member functions points to the base class, not the derived class. You could type cast from the base class to the derived class, but this is a bad practice and would give you access to only the public section of the derived class, not the private section. The result is that you

are forced to use virtual functions for all member functions.

*Problem three:* Two class declarations. Data hiding requires an abstract base class declaration and a derived class declaration that are very similar, but not identical. And all this duplication just because we wanted data hiding.

*Problem four:* Inheritance is disabled. To use inheritance on the class in which data is hidden, you need access to the derived class. But you have access to only the abstract base class declared in an include file, not the derived class declared in the source file that implements the derived class. If you inherit from the abstract base class, you lose the implementation. If you inherit from the derived class, you lose the data hiding.

The bottom line is that implementing data hiding in C++ by using abstract classes disables other advanced features of C++ and adds execution overhead. To use data hiding, you give up inheritance. To use inheritance, you give up data hiding. Data hiding through abstract classes and inheritance do not coexist. This is why I disagree with Stroustrup.

I feel that using abstract classes to perform data hiding in C++ is an afterthought (abstract classes were not added until C++ version 2.0) and a weak solution to an underlying C++ problem that complete data hiding is not built into the language. However, this underlying problem is also a major strength when it comes to execution speed and standard C structure layout compatibility.

In §10.1c of [The Annotated C++ Reference Manual](#), Ellis and Stroustrup hint at a solution (a level of indirection) but dismisses it due to the resulting code being "both larger and slower." Too bad. Those people that want complete data hiding now have to implement it manually.

## 4.9 Chapter Summary

- The class methodology solves the information overload problem by moving data declarations out of include files and into a module, where the data declaration is turned into a private class object.

- There is a fundamental shift away from public access to the data to private access through calling a method function. This object model supports an unlimited number of dynamically allocated objects and object handles that are type checkable by the compiler. A handle is simply a pointer to the object.
- With the class data declaration hidden away in one source file, how can other source files use handles to this class when the class declaration is not even visible? How are these handles type checked by the compiler? The breakthrough in accomplishing this feat is to use an incomplete type. This declares a pointer to a structure tag and allows full usage and type checking of the handle/pointer in other source files. Then, in the implementation module, a data declaration with the same structure tag is declared. This binds the handle to the data declaration and allows handles to be dereferenced only in this one module.
- To dramatically reduce bugs, an object system must provide a means of type checking objects at run-time. An object's type identifier is simply a pointer to a class descriptor structure.
- To avoid changing the `sizeof()` a class declaration by including type information, the underlying heap manager is improved to support run-time type checking.
- The run-time object verification macro, `VERIFY()`, supports a fault-tolerant syntax. It allows a block of code to be executed if and only if the handle the block of code relies on is valid.
- The `NEWOBJ()` and `FREE()` macros hide the programmer from how classes are implemented.
- Class objects should be considered state machines. Method functions then transition the state machine from one valid state to another valid state.



# Chapter 6: Designing Modules [Writing Bug-Free C Code](#)

[6.1 Small versus Large Projects](#)

[6.2 The Key](#)

[6.3 What Is a Module?](#)

[6.4 Designing a Class Implementation Module](#)

[6.5 The USE\\_\\* Include File Model](#)

[6.6 Coding the Module](#)

[6.7 A Sample Module](#)

[6.8 Chapter Summary](#)

What is a module? Is it simply a source file that contains functions? In a high-level sense, yes, but it is also much more. Anyone can write some functions, place them in a file and call them a module. For me, a module is the result of applying a well-founded set of techniques to solving a problem.

## 6.1 Small versus Large Projects

How are small projects usually designed? It has been my experience that small projects are usually written by one person over the course of several hours to several days. In this environment, there is no real concern given to splitting the project into several well-defined source files. More than likely, the small project ends up simply being one source file.

For small projects, this one source file design works quite well, but what about large projects with hundreds of thousands of lines of code and hundreds of source files?

The challenge in working on any project is applying a well founded set of techniques from day one, because every project, no matter how big it is today, started out as a small project.

## 6.2 The Key

The key to successfully coding a hierarchy of modules is that you must always code what to do, not how to do it.

In other words, implement the solution to a problem once, not multiple times. A simple, but effective example is copying a string from one location to another. Do you use `while (*pDst++=*pSrc++)`; or do you use `strcpy(pDst, pSrc)`;? This may seem like an absurd example, but study it carefully. The while

loop is specifying how to copy a string from one location to another, while strcpy() is specifying what to do, leaving the how to strcpy().

The key is to always code *what* to do, not *how* to do it.

This point is so key that I will repeat it. The while loop is specifying how to copy a string from one location to another, while the strcpy() is specifying what to do, leaving the how to strcpy().

I chose strcpy() on purpose because I do not know anyone who would argue for using the while loop instead of strcpy(). Why is the choice of which one to use so obvious?

More than likely, some of the code you have written contains code fragments that specify how to do something instead of specifying what to do.

### 6.2.1 An Example

Let me give you a prime example that comes straight from programming in a GUI environment. In a dialog box, there exists an edit control that contains text that the user can edit, but suppose you want to limit the number of characters that the user can type. How do you do it? In Microsoft Windows, you send a message to the edit control, informing it of the text limit. This is done with the SendDlgItemMessage() function and the EM\_LIMITTEXT message.

```
Limiting text size in a Microsoft Windows edit control  
SendDlgItemMessage( hDlg, nID, EM_LIMITTEXT, wSize, 0L );
```

So, whenever you want to limit the size of an edit control, you call SendDlgItemMessage(), right? I do not think so!

The problem is that by calling SendDlgItemMessage() directly, you are specifying how to limit the text size.

The solution is to code a new function, let's call it EmLimitText(), that calls SendDlgItemMessage(). Whenever you want to limit the text size of an edit

control, you call `EmLimitText()` instead of `SendDlgItemMessage()`.

#### **EmLimitText function**

```
void APIENTRY EmLimitText( HWND hDlg, int nID, WORD wSize )
{
    SendDlgItemMessage( hDlg, nID, EM_LIMITTEXT, wSize, 0L );
} /* EmLimitText */
```

By doing this, you now have the basis for an entire module. For all messages that can be sent to edit controls, provide code wrappers that specify what to do and let the functions call `SendDlgItemMessage()`, which specify how to do it.

### **6.2.2 Another Example**

The class methodology is a prime example of specifying what to do and not how to do it. Consider what coding is like without the class methodology. It is up to each and every module to directly access a data object to perform whatever action is necessary. By directly accessing the object, the code is specifying how to perform the action.

The class methodology forces modules to use method functions to perform an action upon the object. The code is specifying what to do, leaving the how up to the method function.

### **6.2.3 The Advantages**

By far the single biggest advantage of using the technique of specifying what to do instead of how to do it is that it allows you to radically change the how (the implementation), without having to change the what (the function calls).

Another advantage of this technique is that it allows you to make changes to the implementation and recompile only one source file. Since the function interface remains the same, no other source files have to be recompiled.

The turnaround time in making a change and testing it are also dramatically reduced because only one source file needs to be recompiled instead of tens or hundreds.

Another benefit of this technique is that code size is reduced. Instead of repeatedly spelling out in code how to do something, you are now making a function call, which in most cases reduces the code size.

#### **6.2.4 The Goal**

The goal in using this technique in your programs is to reach the point at which implementing new functionality is simply a matter of making a few function calls. This is obviously ideal and does not happen all the time, but the more this technique is used, the more frequently it starts to happen.

#### **6.2.5 Conclusion**

The key to this technique is to always code what to do, not how to do it. In other words, you never want to reinvent the wheel. Instead, implement it once and be done with it! Be careful, however, that you design a solid interface that stands up to the test of time. You do not want to change the interface later, since this requires all modules that use the interface to change as well.

This technique is easy to use when the implementation of something is hundreds of lines long and the obvious choice is to make a function call. However, the real power of this technique comes when it is used extensively in a project, even for implementations that are several lines to one line long. It takes a lot of discipline to use the technique in these cases, but the payoff comes the next time when all you need is a function call.

### **6.3 What Is a Module?**

A module is a source file with well-defined entry points, or method functions (methods), that can be called by other modules. It is the result of applying a well-founded set of techniques to solving a problem. There are primarily two types of modules: code wrapper modules and class implementation modules. Some code wrapper modules also implement a class and hence are considered class implementation modules as well.

Code wrapper modules provide functions that are primarily code wrappers. Most code wrapper functions are independent and stand on their own. Because of this, they can be moved to another module without any compilation

problems. Functions are grouped in a module because they provide similar functionality.

An example is the heap manager module. It sits on top of C's memory allocation routines to provide a cleaner, more robust interface into the system.

Another example is a module that provides code wrappers around all messages that can be sent to edit controls in a GUI environment. An example is using `EmLimitText()` instead of calling `SendDlgItemMessage()` directly.

## **6.4 Designing a Class Implementation Module**

Class implementation modules provide functions that implement a class. The functions must be present in the module for a successful compilation. Functions are grouped in a module because the functions, as a whole, implement the class.

An example of a class implementation module is the random number generator discussed in [§4.7](#).

### **6.4.1 The Interface or API**

By far the toughest and most important part of designing a module is designing the API (Application Programmer's Interface) to the new module.

Once an API is chosen and implemented, it is not likely to change much over time. This is because as a module gets used more and more by other modules, a change in the API is expensive in terms of the time and effort required to implement and debug the change.

An API must be designed right the first time because changes to an API are costly.
--

The first step in creating a new class implementation module API is deciding upon a handle name that is used to refer to objects created by the API. The handle name must begin with an H and be in uppercase. A name is chosen so that it is obvious, unique, relatively short and has a good mixed case name, usually spelled the same as the handle. You also need a module name so that

functions of the module follow the module/verb/noun naming convention.

In the random number generator, HRAND was chosen as the handle name. It is pretty obvious, unique from all other handle names and relatively short. The variable name used to refer to objects of the class is hRand. Rand is the module name used to prefix method functions, as in RandCreate() and RandDestroy().

Suppose we are creating a class module that provides a code wrapper around opening, reading, writing and closing operating system files. In this case, the code wrapper module is a class implementation module. I would choose a handle name of HDOSFH (Handle to the DOS File Handle), a variable name of hDosFh, a module name of Dos and method function names of DosOpenFile(), DosRead(), DosWrite() and DosCloseFile().

All modules that implement a class have at least two method functions: one method function that creates the object (for example, RandCreate() and DosOpenFile()) and another method function that destroys the created object (for example, RandDestroy() and DosCloseFile()).

An important concept of class modules is that the method functions act upon dynamically allocated objects, objects that are created and destroyed using method functions. The method functions do not act upon static objects.

#### **6.4.2 An Interface Specification Template**

The interface specification for new class modules follows a general template. First of all there is a NEWHANDLE(HOBJ) declaration that is at the top of a global include file. This adds a new type checkable data type, named HOBJ, into the system. It is declared near the top of the include file, next to all other NEWHANDLE() declarations, so that, if needed, HOBJ may be used in the prototypes of other USE\_ sections.

The NEWHANDLE(HOBJ) declaration is not included in the USE\_HOBJ section because to use only the HOBJ data type in other USE\_ prototype sections would require the entire USE\_HOBJ section to be included first. Why include the entire section when all that is needed is access to NEWHANDLE(HOBJ)?



#### Template interface for a new module

```
NEWHANDLE (HOBJ);  
  
.  
.  
.  
#ifdef USE_HOBJ  
/*-----  
*  
*   Short one line description of module  
*  
*-----*/  
EXTERNC HOBJ APIENTRY ObjCreate      ( arguments );  
EXTERNC type APIENTRY ObjMethodName1 ( HOBJ, arguments );  
EXTERNC type APIENTRY ObjMethodName2 ( HOBJ, arguments );  
...  
EXTERNC type APIENTRY ObjMethodNameN ( HOBJ, arguments );  
EXTERNC HOBJ APIENTRY ObjDestroy     ( HOBJ );  
#endif
```

The function prototypes for the method functions of the class appear later in the include file. A create method that returns a handle to the created object is present along with all other method functions for the class (which expect the object handle as the first argument). By convention, NULL is always returned by the ObjDestroy() method. The arguments and return type of all other method functions depend upon the class implementation. Usage of APIENTRY is discussed later in this chapter.

Notice that the module prototypes are enclosed within an #ifdef / #endif section. This implements the USE\_\* include file model, which is discussed next.

## 6.5 The USE\_\* Include File Model

When I first started coding in C, I followed the traditional technique of placing data declarations and function prototypes in a separate include file and explicitly including it in whatever source files needed access to the information.

This traditional technique works for small projects, but it does not scale to large projects well, because before you know it, you end up with a lot of include files and a lot of interdependencies between them. To avoid headaches you end up

including almost every include file in every source file. At least that is what happened to me. Also, compilation times got longer and longer.

Using the [class mechanism described in Chapter 4](#) eliminated the data interdependency problem, but it still left me with long compile-times and a lot of include files. How could I eliminate all the include files and speed up the compile-times?

As the project I was working on got larger and larger, I began to notice something interesting. The include file model for a small project and a large project are different.

A small project tends to have a few source files with a lot of functions that perform a lot of varied tasks. This requires a lot of include files in each source file.

A large project tends to have highly specialized modules that contain a few method functions and a few internal support functions which implement a specific class object. This requires just a few include files in each source file.

The include file model for a small project and a large project are different.
---

The solution to all my problems came in the form of one global include file with all but a few sections enclosed in `#ifdef USE_HOBJ / #endif` sections. At the top of the global include file are all `NEWHANDLE()` declarations and function prototypes to commonly used system level code. Within each `#ifdef USE_HOBJ / #endif` section are the prototypes for the `HOBJ` class, where `HOBJ` is a place holder for the name of the class.

At the top of every module is a list of `#define`'s enumerating what that module uses. Following this is a `#include` of the global include file.

The combination of the class mechanism and the new `USE_*` model caused the compilation times to improve dramatically and reduced the number of include files to just one.

### 6.5.1 Precompiled Headers Surprise



By now, many of you are probably wondering why precompiled headers were not used instead? Well, I tried them. They increased the build time of my large project by 50% percent. I was surprised, but the reasons make sense.

My project is huge with all modules being specialized. Because of this, each module is really not including that much because of the `USE_*` include model. However, the precompiled header that was being used contained the include information that was needed by all modules. The precompiled header was huge, but the compiler was able to load it fast and start compiling the source file right away. So the extra build time was not due to the huge precompiled header size. Then why was the build time of the project 50 percent longer?

My best guess is that the huge precompiled header was causing longer symbol table search times within the compiler. Without the precompiled header, only a few sections of the include file were being included, causing the compiler symbol table to be practically empty and short symbol table search times.

## 6.6 Coding the Module

Now that the module interface has been designed, it is time to start coding the module. It is not enough that a module be coded bug-free. It must also look good and be documented well. The true test of a well-written module is if your coworkers can take that module and read it, understanding everything that is going on without any help from you.

What looks good is highly subjective. However, I highly recommend that you pick some documenting style that works for you and subject the style to a review by your coworkers. After all, you have to read and modify their code and they have to read and modify your code!

### 6.6.1 The Copyright Header

At the top of every module (source file), there should be a copyright notice. Where I work, it looks something like the following.

```
The copyright header
/*****
/*
/*
/*      (project name)
/*
```

```

/*                                                                    */
/*    Copyright (date) (Company Name). All rights reserved.          */
/*                                                                    */
/*    This program contains the confidential trade secret             */
/*    information of (Company Name). Use, disclosure, or              */
/*    copying without written consent is strictly prohibited.        */
/*                                                                    */
/******

```

(Company Name), (project name) and (date) are place holders to be filled in by you.

## 6.6.2 Module Documentation

Following the copyright header is a comment section that describes the module as a whole.

### The module comment header

```

/*pm-----
-
*
*  OUTLINE:
*
*      (Describes the purpose of the module)
*
*  IMPLEMENTATION:
*
*      (Describes in high level terms how the module works)
*
*  NOTES:
*
*      (Enumerate noteworthy items)
*
*-----
*/

```

The OUTLINE section. Use this section to describe why the module needs to exist. What is it doing? Pretend that a coworker walked up to you and asked what you are working on.

The IMPLEMENTATION section. Use this section to spell out the major algorithms that you are going to use to implement the module. Again, pretend

that a coworker asked you how you are going to implement the module that you just described to him or her.

The NOTES section. This section is a catchall section where you can put anything you want. I use this section for notes that would be helpful to anyone who has to modify the code months down the road. Another use is to document special situations that must be tested before the modified code can be checked back into a version control system.

Usage of pm in the comment header is used by an [automatic documentation tool](#).

### 6.6.3 Include File Section and USEWINASSERT

Following the module documentation is a series of #define USE\_'s followed by the #include of the global include file and USEWINASSERT.

```
Include section example  
#define USE_HRAND  
#define USE_HDOSFH  
#include "app.h"  
USEWINASSERT
```

A module always has at least one #define USE\_\*, because you always want #included the section of the include file for the module you are working on.

### 6.6.4 The Class Declaration

What follows next is usually a class declaration for the object that is being implemented by the module. The class declaration for the random number generator looks like the following.

```
Random number generator class declaration  
CLASS(hRand, HRAND) {  
    long lRand;  
};
```

## 6.6.5 Prototypes of LOCAL Functions

Following the class declaration are the prototypes for functions that are used and defined only in this module. It is important for proper error checking by the compiler that every function be prototyped before being used or called.

Every function in the module must be prototyped.

The method functions of the module are prototyped in the global include file and the proper `#define USE_*` causes them to be included. The functions that are local to this module must not be prototyped in the global include file because they are not part of the module interface that is called by other modules. Instead, they are private to the module and prototyped in the module.

## 6.6.6 APIENTRY (Method) Functions

I usually organize my module files so that all the functions that are entry points into the module appear at the top of the source file and all local functions appear after the entry point functions.

It is important to provide a comment header for every single function in the module that properly documents the functions. Where I work, a comment header that looks like the following is used.

### The function comment header template

```
/*pf-----  
-  
*  
* DESCRIPTION: (A few word description) initials  
*  
* (A long description of the function)  
*  
* ARGUMENTS:  
*  
* Arg1 - Arg1 description  
* ...  
* ArgN - ArgN description  
*  
* RETURNS:  
*  
* (A description of the return value)
```

```

*
*  NOTES:
*
*      (optional notes section)
*
*-----
*/

```

The DESCRIPTION section. The section starts off with a terse description of the function in parentheses. Following this are the initials of the programmer(s) who worked on this function. The body of this section contains a sentence or two that describe what the function does.

The ARGUMENTS section. This section spells out the arguments that the function takes. There is one line per argument. The name of the argument is listed, followed by a dash and a short description of the argument.

The RETURNS section. This section describes the value that is returned by the function. If there is none, place (void) here.

The NOTES section. The notes section is optional and does not appear in all function comment headers. When present, it serves the same purpose as the notes section in the module comment header. I use this section to leave notes that would be helpful to anyone who has to modify the code months down the road.

Pf in the comment header is used by an [automatic documentation tool](#).

Following the comment header is the entry point function itself. The template for an entry point function looks like the following.

```

Template for entry point function
return-type APIENTRY FunctionName( type arg1, ..., type argN )
{
    (function body, usually in fault-tolerant form)
} /* FunctionName */

```

Notice that the new-style standard C form of declaring the argument list is

used. Also, following the ending brace of the function is a comment containing the name of the function the end brace belongs to. The only thing remaining is the usage of APIENTRY.

APIENTRY is a macro that is used to assign attributes to entry point functions. The important thing to remember is that APIENTRY is used to present a logical view to the programmer. The actual implementation of APIENTRY varies from environment to environment.

```
The APIENTRY define  
#define APIENTRY FAR PASCAL
```

For example, if you are programming under the Intel segmented architecture and using Microsoft's C8 compiler, FAR and PASCAL are actually defined to be something (see [§3.2.1](#)). Due to the segmented architecture and the possibility of near or far code, APIENTRY functions must be accessible to other modules and hence, declared as FAR. Using PASCAL is optional but provides a savings in code size due to how arguments are pushed and popped (see [§2.1.11](#)).

If you are programming in a 32-bit flat model environment, FAR and PASCAL are defined to be nothing, so the function is public and accessible to other modules, which is the desired behavior.

```
The APIENTRY define, specific to Microsoft C8 Windows DLL  
programming  
#define APIENTRY EXPORT FAR PASCAL
```

Finally, if you are programming under Microsoft Windows and writing a DLL, you want to ensure that your API functions are exported. This is done with the EXPORT keyword.

APIENTRY is specifying what to do, not how to do it. The how is left to a macro that can be easily changed at any time. This also allows for a single code base that can be targeted to multiple platforms without any code changes.

### 6.6.7 LOCAL Functions

In the process of implementing modules I believe you will quickly find out that it is not always possible or desirable to fully implement an APIENTRY function in one function. You will end up calling support or helper functions that are private to the module you are working on.

The template for a LOCAL function is almost identical to an APIENTRY function.

**Template for a LOCAL function**

```
return-type LOCAL FunctionName( type arg1, ..., type argN )
{
    (function body, usually in fault-tolerant form)
} /* FunctionName */
```

The comment header and body style are the same except that APIENTRY has been replaced by LOCAL.

Because LOCAL functions are intended to be private functions, callable only by the module they are contained in, you should try to ensure that they can be called only by the one module. Another potential problem is how to name the LOCAL functions. You do not want to have to worry about name conflicts with LOCAL functions in other modules.

Luckily, C provides a solution to these problems.

If a function is declared as static, it is guaranteed by C that the function is visible only to the source file that declared the static function. What this means is that the function cannot be called from other source files because the function is not even visible to them. You no longer have to worry about name conflicts of LOCAL functions between modules because the LOCAL function names are not even visible to the other modules.

**The LOCAL and LOCALASM define**

```
#define LOCAL static NEAR FASTCALL
#define LOCALASM static NEAR PASCAL
```

If you are programming in a 32-bit flat model environment, LOCAL is defined to be static because NEAR, FASTCALL and PASCAL are all defined to be nothing (see §3.2.1). However, if you are using Microsoft C8, several optimizations can be applied to LOCAL functions.

The first optimization is the usage of NEAR. This tells the compiler that the function is in the same code segment as the caller of the function and that a near (16-bit) function call should be used instead of a far (32-bit) function call. There is a big performance hit when you compare the execution speed of a far call to that of a near call. A far function call can be up to five times as slow as a near function in using Intel 80486 protected-mode.

The second optimization is the usage of FASTCALL. This instructs the compiler to attempt to pass as many arguments as possible to the function through the CPU's registers instead of on the stack.

The LOCALASM define is used for local functions containing assembly code. This is needed for Microsoft C8 because assembly code and the register calling convention are incompatible. Using PASCAL provides a savings in code size due to how arguments are pushed and popped (see §2.1.11).

## 6.7 A Sample Module

What follows is a simple code wrapper around the standard C run-time library open(), read(), write() and close() calls.

### 6.7.1 The Include File

```
HDOSFH include file section
NEWHANDLE (HDOSFH) ;
.
.
.
#ifdef USE_LOWIO
/*-----
*
*   Access to low-level I/O run-time library functions
*
*-----*/
```



```

#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#endif

#ifdef USE_HDOSFH
/*-----
 *
 *   Code wrapper to low-level file I/O
 *
 *-----*/

EXTERNC HDOSFH APIENTRY DosOpenFile  ( LPSTR );
EXTERNC WORD   APIENTRY DosRead      ( HDOSFH, LPVOID, WORD );
EXTERNC WORD   APIENTRY DosWrite     ( HDOSFH, LPVOID, WORD );
EXTERNC HDOSFH APIENTRY DosCloseFile ( HDOSFH );
#endif

```



## 6.7.2 The Module File

### HDOSFH module

```

/*****
/*
/*                                     */
/*      (project name)                                     */
/*
/*      Copyright (date) (Company Name). All rights reserved.  */
/*
/*      This program contains the confidential trade secret    */
/*      information of (Company Name). Use, disclosure, or      */
/*      copying without written consent is strictly prohibited. */
/*
*****/

/*pm-----
 *
 *   OUTLINE:
 *
 *       This module provides access to the low-level file I/O
 *       functions of the standard Microsoft C run-time library.
 *
 *   IMPLEMENTATION:
 *
 *       This module is simply a code wrapper module.
 *
 *   NOTES:

```

```

*
*-----*/

#define USE_LOWIO
#define USE_HDOSFH
#include "app.h"

USEWINASSERT

/*--- The class object ---*/
CLASS(hDosFh, HDOSFH) {
    int fh;
};

/*pf-----
*
*   DESCRIPTION: (Open File)   JLJ
*
*   Attempt to open a file
*
*   ARGUMENTS:
*
*   lpFilename - The name of the file to open
*
*   RETURNS:
*
*   A file object handle or NULL if there was some error
*   in opening the specified file.
*
*-----*/

HDOSFH APIENTRY DosOpenFile( LPSTR lpFilename )
{
    HDOSFH hDosFh=NULL;
    int fh=open(lpFilename, _O_RDWR|_O_BINARY);
    if (fh!=-1) {
        NEWOBJ(hDosFh);
        hDosFh->fh = fh;
    }
    return (hDosFh);
} /* DosOpenFile */

/*pf-----
*
*   DESCRIPTION: (Close File)   JLJ
*
*   Close a previously opened file
*
*   ARGUMENTS:

```

```

*
*   hDosFh - The file object or NULL
*
* RETURNS:
*
*   NULL
*
*-----*/

HDOSFH APIENTRY DosCloseFile( HDOSFH hDosFh )
{
    VERIFYZ(hDosFh) {
        int nResult=close(hDosFh->fh);
        WinAssert(!nResult);
        FREE(hDosFh);
    }
    return (NULL);
} /* DosCloseFile */

/*pf-----
*
* DESCRIPTION: (Read File)   JLJ
*
*   Attempt to read a block of information from a file.
*
* ARGUMENTS:
*
*   hDosFh - The file object
*   lpMem   - Pointer to memory buffer
*   wCount  - Number of bytes to read into the memory buffer
*
* RETURNS:
*
*   The number of bytes that were actually read
*
*-----*/

WORD APIENTRY DosRead( HDOSFH hDosFh, LPVOID lpMem, WORD wCount )
{
    WORD wNumRead=0;
    VERIFY(hDosFh) {
        wNumRead = (WORD)read(hDosFh->fh, lpMem, wCount);
    }
    return (wNumRead);
} /* DosRead */

/*pf-----
*

```

```

*   DESCRIPTION: (Write File)   JLJ
*
*   Attempt to write a block of information to a file.
*
*   ARGUMENTS:
*
*   hDosFh - The file object
*   lpMem  - Pointer to memory buffer
*   wCount - Number of bytes to write to the file
*
*   RETURNS:
*
*   The number of bytes that were actually written
*
*-----*/
WORD APIENTRY DosWrite( HDOSFH hDosFh, LPVOID lpMem, WORD wCount )
{
    WORD wNumWritten=0;
    VERIFY(hDosFh) {
        wNumWritten = (WORD)write(hDosFh->fh, lpMem, wCount);
    }
    return (wNumWritten);
} /* DosWrite */

```

### 6.7.3 Commentary

Use this module as a template on how to write modules. It is bare-bones and targeted to MS-DOS using Microsoft C8, but you should be able to adapt it easily to other environments. I would like to emphasize some parts of this module.

**Completeness.** This module is not complete. There are other low-level I/O functions that should be implemented.

**The includes.** This module is intended to be a code wrapper that totally replaces the run-time library low-level I/O. Therefore, this module should be the only module that needs to do a `#define USE_LOWIO`. Notice that the `#includes` for `USE_LOWIO` are not done in the source file but are instead done in the global include file.

**Accessing low-level I/O.** Whenever any source file wants to access the low-

level I/O, it should now use the code wrapper code and do a `#define USE_HDOSFH`.

`DosCloseFile` uses `VERIFYZ`. It is important that only method functions that destroy an object allow `NULL` to be passed in as an argument. You do not want to trigger a run-time object verification failure. `VERIFYZ` performs this task.

Fault-tolerant methods. Whenever possible, the fault-tolerant form of `VERIFY` should be used. For functions returning a value, a reasonable failure return value is in place before the run-time verification takes place. This way, even if a bad object handle is unknowingly passed in, the calling code will react to the failure value.

## 6.8 Chapter Summary

- The key to successfully coding a hierarchy of modules is that you must always code what to do, not how to do it.
- In this way, you avoid spreading knowledge about how to do something and instead put this knowledge in a function in one place. You are now free to call the function as many times as you want in as many places as you want. Changing the implementation down the road is a lot easier since the implementation is now isolated in one function.
- An important benefit of this technique is that it allows for the rapid prototyping of changes to an implementation, because changing the implementation changes only one source file as opposed to many.

# Chapter 7: General Tips [Writing Bug-Free C Code](#)

[7.1 Design It Right the First Time](#)

[7.2 Good Design Beats an Optimizing Compiler](#)

[7.3 Evolutionary Coding](#)

[7.4 Set Goals](#)

[7.5 Code What to Do, not How to Do It](#)

[7.6 Virtually No Global Variables](#)

[7.7 Loop Variables](#)

[7.8 Use the Highest Compiler Warning Level](#)

[7.9 Use "static" to Localize Knowledge](#)

[7.10 Place Variables in the Block Needed](#)

[7.11 Arrays on the Stack](#)

[7.12 Pointers Contain Valid Addresses or NULL](#)

[7.13 Avoid Type Casting Whenever Possible](#)

[7.14 Use sizeof\(\) on Variables, not Types](#)

[7.15 Avoid Deeply Nested Blocks](#)

[7.16 Keep Functions Small](#)

[7.17 Releasing Debugging Code in the Product](#)

[7.18 Stack Trace Support](#)

[7.19 Functions Have a Single Point of Exit](#)

[7.20 Do Not Use the Goto Statement](#)

[7.21 Write Bulletproof Functions](#)

[7.22 Writing Portable Code](#)

[7.23 Memory Models](#)

[7.24 Automated Testing Procedures](#)

[7.25 Documentation Tools](#)

[7.26 Source-Code Control Systems](#)

[7.27 Monochrome Screen](#)

[7.28 Techniques for Debugging Timing Sensitive Code](#)

No matter how good your tools that help you detect bugs in your programs, the goal of every programmer should be to avoid bugs in the first place. Debugging tools serve a purpose, but relying on the tools to catch your programming mistakes does not make you a better programmer. What makes you a better programmer is learning from your mistakes.

A key to writing bug-free code is learning from your mistakes.

How do you rate your own ability to produce bug-free code? Whatever your answer is, how did you rate yourself last year? How do you expect to rate yourself next year?

The point of this exercise is to stress to you the importance of improving your coding techniques year after year. The goal is to be able to write more code next year than last year and to write it with fewer bugs.

## 7.1 Design It Right the First Time

If there is one thing that I have learned over the years that I would emphasize to a programmer just starting out, it is that old code rarely dies because it just stays around and around. This is especially true in large projects. And all successful small projects end up turning into large projects.

As time goes on in a large project, more and more code layers are built upon existing code layers. It does not take long before coding is no longer being done to the operating system level or windowing environment layer but to the layers that were coded on top of these system layers.

Now suppose that a year later someone discovers that the implementation of a low-level layer is causing performance problems. All too often management decides to live with the performance problem in favor of using their programmer's time putting new features into the product. You are forced by management to live with a lot of the coding decisions you make over the years, so your decisions had better be good!

Management may allow you to re-engineer a module to make it better and faster, but consider the lost time. The time that is attributed to the module is the original design time plus the time to re-engineer the module. Designing a module right the first time saves time.

It is not feasible to continually re-engineer old modules. If this does happen due to poor design decisions made upfront, a project will come to a halt.

Designing a module right the first time saves time.
---

My advice is to design a module right the first time because you'll rarely get the chance to re-engineer the module.

## **7.2 Good Design Beats an Optimizing Compiler**

An optimizing compiler helps a program run faster, but in all cases, a good design makes a program run fast. A great design produces the fastest program. A little more time spent on a programming problem generally results in a better design, which can make a program run significantly

faster.

Over the last few years, I think all of us at one time or another saw a well-known product upgrade getting hammered by the trade press for how sluggishly the upgrade performed. Do not let this happen to your product.

And by all means, evaluate your competition. If your product is two to three times slower than the competition and this comes out in a magazine review, do you think potential customers will buy your product?

## **7.3 Evolutionary Coding**

How do you code a module? Do you spend days working feverishly on a module and have it all come together that last half day? Do you code one piece of a module, moving on to the next piece only when you are certain that the piece you just wrote is working?

Over the years I've tried both techniques and I can tell you from experience that coding a module a piece at a time is much easier. It also helps isolate bugs. If you wait until the end to put all the pieces together, where is the problem? By coding one piece at a time, the problem is more than likely in the piece you are working on and not in some piece you have already finished and tested.

### **7.3.1 Build a Framework**

Start coding a module by building a framework that is plugged into a system almost immediately. The goal in creating this framework is to get the module completely stubbed out so that it compiles.

Let's use the Dos module as an example and assume that the module interface has already been designed. The first step in creating the framework is to create the global include file. Next, create a source file that contains all of the sections of a module, but none of the guts. Namely, create the module and APIENTRY comment blocks without any comments, declare the class without any members and write all the APIENTRY functions with no code in the function bodies.



At this point, I place return statements in all functions, so that each function can return an error value. For example, I have `DosOpenFile()` and `DosCloseFile()` return NULL and `DosRead()` and `DosWrite()` return zero.

The framework is now complete. Compile the module and correct any errors that show up. Although no real code has been written, the framework provides you with a clear goal.

### **7.3.2 Code the Create and Destroy Methods**

After the framework is compiling successfully, you are ready to start implementing the module. Where should you start? The functions that I always tackle first are the create and destroy methods. This allows me to write some simple code that uses and tests the module.

In the case of the Dos module, the `DosOpenFile()` and `DosCloseFile()` functions are implemented first, followed by some test code. This test code allows verification that (1) opening an existing file works, (2) trying to open a file that does not exist fails and (3) calling `DosCloseFile()` works properly and frees allocated memory.

Most modules are not as simple as the Dos module was in determining the data items to add to the class structure. This is fine. Simply fill in the class structure with whatever data items are required by the create and destroy method function. When implementing other method functions, add data items to the class structure as they are needed.

### **7.3.3 Code the Other Methods**

Once the create and destroy methods are working correctly, it is time to start implementing the other method functions. The best strategy is to implement a method function, compile it and then test it by writing test code.

In the case of the Dos module, you may decide to implement `DosRead()` first. After doing so, you could then create a test file with a known data set and attempt to have the test code read this data set. This helps validate the

DosRead() code. Likewise for the DosWrite() function. Write out some information to a file and verify that it was indeed written.

The order in which you code the method functions is totally up to you. After you gain experience using this technique, you will end up picking an order that allows for test code to be written easily.

## **7.4 Set Goals**

Goal setting is always important to accomplishing any task you set out to do, but it is especially important that programmers have clearly defined goals. The problem is that all too often a programmer gets sucked into a project that is 90 percent done and weeks or months later the project is still only 90 percent done.

### **7.4.1 The 90 Percent Done Syndrome**

The problem with programming without a clear well-defined goal is that a project is not completed.

Programming without a goal is like a sailboat without a sail. You drift.
--

I have fallen into this trap myself. You work on items that relate to the ninety percent that is already done. You may have found a new feature to add and so you work on it, but the feature does not help you complete the project. Or you have found a better way to implement an algorithm, so you spend time recoding the algorithm. While finding a better algorithm is great, it only delays the project from being completed.

The module framework helps you set a clearly defined goal. Once the framework is in place, all the APIENTRY functions are stubbed out, just waiting to be completed. Once these APIENTRY functions are completely written and tested, you have reached your goal and the module is finished.

## **7.5 Code What to Do, not How to Do It**

The goal of this technique is to avoid spreading knowledge about how something is done throughout the entire project. By moving this knowledge to one function, you are isolating the knowledge. Now any code that calls this function is specifying what must be done, leaving the how to the function.

Never reinvent the wheel. Always code what to do, not how to do it.

It is hard to discipline yourself to use this technique, but the payback is well worth the extra effort.

Code that uses this technique becomes shorter and more compact. What used to take ten lines now takes five. This leads to a program that is easier to code, maintain and change.

A good analogy is building blocks. You are building something starting from scratch so you start out by making a few blocks. You use these building blocks to make the something. Now let's say that you want to build something else. The key is that you don't start totally from scratch because you've already built some of the basic building blocks.

Always keep your eyes open for a new building block to implement.

## **7.6 Virtually No Global Variables**

In an object-based system, global variables (variables known to more than one source file) are almost never needed. The reasoning is simple. Objects, by design, are created and destroyed by method functions, so the objects themselves are never static. An advantage of object-based systems that dynamically allocate objects is that they are never limited by some predefined number. Instead, they are limited only by the amount of memory that is available.

Global variables are rarely referenced by the method functions of an object because method functions act upon an object, not global data. Remember that an object's handle is passed in as the first argument to a method function.

So why are global variables ever needed? Sometimes to directly support a class of objects. A prime example is if all objects of a class require access to a read-only resource. When creating the first object for this class, the resource is read into memory and a handle to the resource is stored in a global variable. When destroying the last object for this class, the memory associated with the resource is freed.

Global variables to support individual object instances should not be allowed. Global variables that support a class of objects as a whole are permitted. Because these global variables support a class, their use is limited to the source file that declares the class. This can be enforced by using static on these variables, ensuring that the variables are not visible in other source files.

## **7.7 Loop Variables**

Variables used to control for loops should not be used outside the loop itself. The LOOP() macro (see [§2.2.5](#)) was designed to enforce this rule.

Loop variables should not be used outside the loop itself.
--

This rule makes code easier to maintain. While you are writing a function, you understand it thoroughly and are unlikely to make a mistake in using the loop variable after the loop has exited. However, when you or one of your coworkers modifies the loop next year, the assumptions under which the looping variable is used could easily be invalidated.

It is better to be safe than sorry.

## **7.8 Use the Highest Compiler Warning Level**

Today's compilers are pretty smart, but only if you tell them to be. By default, most compilers use a low warning level and use more advanced error checking only if instructed to do so. This is almost always done to remain compatible with older code. You can imagine the number of support calls a compiler vendor would get if the old code that used to compile fine

all of a sudden started producing a lot of warning messages under a new release of the compiler.

 For more information on setting the compiler warning level in Microsoft C8, see [§2.1.3](#).

## 7.9 Use "static" to Localize Knowledge

Using static can sometimes be confusing because it appears to mean different things in different contexts. A simple rule that helps clear things up is that using static assigns permanent storage to an object and limits the visibility of the object to the scope in which it is defined. There are three basic ways in which static can be used.

Static function declaration. You have already seen static used in this case with the LOCAL macro (see [§6.6.7](#)). Since a function already has permanent storage, this part of the rule is redundant. The scope of a function is file scope, so static limits the visibility of the function to the rest of the source file in which it is defined.

External static variables. When a variable is defined at file scope, it already has permanent storage and is visible to all source files. Using static makes the variable invisible to other source files. Again, the variable definition already has permanent storage, so this part of the rule is redundant.

Internal static variables. When a variable is defined within a function, it is called an automatic variable. Automatic variables have no persistence across function call invocations. In other words, if a value is assigned to an automatic variable in a block, that value is lost when the block is exited. In almost all cases, this is the desired behavior. However, using static assigns permanent storage to the variable so that a value assigned to the variable is not lost. The scope of the variable is also limited to the block in which it is defined. An internal static variable is just like an external static variable except that the visibility of the variable is limited to a block.

It is also important to understand how an initialized static variable behaves.

Consider the following code.

**An example in using static**

```
void Testing( void )
{
    LOOP(3) {
        static int nTest1=100;
        int nTest2=100;
        printf( "nTest1=%d, nTest2=%d\n", nTest1, nTest2 );
        ++nTest1;
        ++nTest2;
    } ENDLLOOP
/* Testing */
```

**Output from calling Testing() three times**

```
nTest1=100, nTest2=100
nTest1=101, nTest2=100
nTest1=102, nTest2=100
nTest1=103, nTest2=100
nTest1=104, nTest2=100
nTest1=105, nTest2=100
nTest1=106, nTest2=100
nTest1=107, nTest2=100
nTest1=108, nTest2=100
```

As you can see from this example, nTest1 is initialized only once, while nTest2 is initialized on each pass through the block. The rule is that the initialization of any static variables takes place at compile-time and that the initialization of automatic variables takes place at run-time.

Static variables are initialized once at compile-time. Automatic variables are initialized as needed at run-time.

## 7.10 Place Variables in the Block Needed

How many times have you tracked down a bug only to realize that you used an automatic variable before it was properly initialized, or used the variable well after it should have been, when it no longer contained an appropriate

value? While this may not happen to you as you write the function, it becomes a lot more likely when you go back to the code at a later date and modify it.

The solution to this problem is to define variables only in the innermost scope in which they are needed. A new scope is created any time you use a begin brace {. The scope is terminated by an ending brace }. This means that your if statements, while statements, and so on, create new scopes. Variables defined within a scope are visible only within that scope. As soon as the scope ends, so does your access to the variable. Consider the following code fragment.

```
Limiting the scope of a variable  
LOOP(strlen(pString)) {  
    char c=pString[loop];  
    ...  
} ENLOOP
```

In this example, c is visible only within the loop. As soon as the loop exits, c is longer visible. In fact, it can be defined and reused in another scope.

**Define variables in the scope in which they are needed.**

In standard C, variables can be defined only at the beginning of a scope before the main body of code. In C++, variables can be defined wherever a statement is valid.

A useful #define for both C and C++ is the NewScope define.

```
NewScope define  
#define NewScope
```

NewScope is a syntactical place holder (defined to be nothing) that allows a new scope to be introduced into a program. It also allows for a natural

indentation style.

#### **Using NewScope**

```
void APIENTRY Function( args )
{
    /*--- Comment ---*/
    (code block)
    /*--- Using NewScope ---*/
    NewScope {
        type var;
        (code block that uses var)
    }

} /* Function */
```

NewScope is useful in both C and C++ because it allows variables to be created that are private to a block. As soon as the block exits, the variables declared in the block are no longer visible and cannot be referenced.

## **7.11 Arrays on the Stack**

When using arrays that are declared on the stack, you must be careful not to return a pointer to one of these arrays back to the calling function. Consider the following example.

#### **A program with a subtle bug**

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

char *myitoa( int nNumber )
{
    char buffer[80];
    sprintf( buffer, "%d", nNumber );
    return (buffer);
}

int main(void)
{
    printf( "Number = %s\n", myitoa(234) );
}
```



```
    return 0;  
}
```

This program contains a subtle bug in that `myitoa()` is returning the address of `buffer`, an array on the stack. It is subtle because despite this bug, the program still works properly!

It is a bug to return the address of an array on the stack back to a calling function because after the function returns, the array is now in the part of the stack that no function owns. The problem with this bug is that the code will work unless the array gets overwritten by making another function call, although, making a function call is no guarantee that `buffer` will be overwritten.

This bug is exactly like keeping a pointer around a block of memory that was freed. Unless the memory is reused and overwritten, accessing an object through the memory pointer will continue to work. As soon as the block of memory gets overwritten, you have a subtle problem to track down.

## 7.12 Pointers Contain Valid Addresses or NULL

The reasoning behind this is simple. It helps find bugs. If all pointers contain either a valid address or NULL, then accessing the valid address does not cause the program to fault. However, accessing the NULL pointer under most protected environments causes the program to fault and you have found your bug.

Consider what would happen if you have a pointer to a memory object, the memory object is freed and the pointer is not set to NULL. The pointer still contains the old address, an address to an invalid object. However, the address itself under most environments is still valid and accessing the memory does not cause your program to fault. This is a problem and the source of a lot of bugs.

The reason for being able to access the memory of a freed object is that most memory managers simply add the memory back into the pool of free

memory. Rarely do they actually mark this memory as being inaccessible to the program. It is time-consuming to communicate with the operating system on every allocation and deallocation request. Instead, a memory pool is maintained. Only when this pool is exhausted does the heap manager ask the operating system for more memory.

To help enforce this policy, macros that interface with the memory manager should be designed and used exclusively. An example of this is `NEWOBJ()` or `FREE()` for class objects (see [§4.5](#)).

## 7.13 Avoid Type Casting Whenever Possible

Type casting, by design, bypasses the type checking of the compiler. In essence, type casting tells the compiler that you know what you are doing and not to complain about it. The problem with this is that your environment may change, you may be porting the software to another platform, changing the memory model of the program or upgrading to a new revision of the compiler.

For whatever reason, your environment has changed. When you recompile your program, you run the risk of missing warning messages. This is because the behavior of the statement in which you are using the type cast may have changed, but the type cast masks the behavior change.



There is another situation that is especially true in mixed-model programming under Windows. To make matters worse, Microsoft's own sample code is a bad example because it is littered with totally unnecessary type casts. Consider the following code fragment.

### **An example of bad type casts**

```
MSG msg;
while (GetMessage((LPMSG)&msg, NULL, 0, 0)) {
    TranslateMessage((LPMSG)&msg);
    DispatchMessage((LPMSG)&msg);
}
```

Microsoft Windows programmers recognize this code as the main message loop for an application. All messages that are placed in an application's message queue are dispatched by this message loop.

The problem with this code is that all three type casts to LPMSG are totally unnecessary. The code works great without the type casts. The prototypes for GetMessage(), TranslateMessage() and DispatchMessage() all indicate that they take LPMSG as an argument. The data type of &msg is PMSG due to the mixed-model environment. I can only suppose that the programmer thought that PMSG must be type cast into what the functions expected, an LPMSG. This is simply not the case. In mixed-model programming, the compiler promotes a near pointer to the object to a far pointer to the object in all cases that a far pointer to the object is expected. In other words, the compiler is implicitly performing the type cast for you.



### 7.13.1 Mixed-Model Programming Implicit Type Cast Warning

In mixed-model programming there exists a subtle problem if you write code that allows NULL to be passed through as one of the argument pointers. Consider the following code.

```
GetCpuSpeed(), demonstrating implicit type cast problem
int APIENTRY GetCpuSpeed( LPSTR lpBuffer )
{
    int nSpeed=(calculation);
    if (lpBuffer) {
        (fill buffer with text description of speed)
    }
    return (nSpeed);
} /* GetCpuSpeed */
```

GetCpuSpeed() always returns the CPU speed as an int, but as an option it also creates a text description of the CPU speed in the provided buffer if the buffer pointer is non-NULL. Now what happens when you call

## GetCpuSpeed()?

### **Calling GetCpuSpeed()**

```
PSTR pBuffer=NULL;
int nSpeed1=GetCpuSpeed(NULL);
int nSpeed2=GetCpuSpeed(pBuffer);
```

In both cases you want only the integer speed and not the text description. In the first case, `GetCpuSpeed(NULL)` behaves as expected. However, in the second case, `GetCpuSpeed(pBuffer)` fills in a text buffer. The problem is that `pBuffer` is a near pointer and that `GetCpuSpeed()` expects a far pointer. No matter what value is contained in `pBuffer`, it is considered a valid pointer and the type cast of a near pointer to a far pointer (implicitly by the compiler or explicitly by you) uses the data segment value as the segment for the far pointer.

In other words, when the `pBuffer` near pointer is converted to a far pointer, the offset is `NULL`, but the segment value is non-`NULL`.

From experience, I have found that correctly writing code in situations like this is too problematic. My solution to this problem has been to move away from mixed-model pointers and stick with far pointers.

## 7.14 Use `sizeof()` on Variables, not Types

How do you use `sizeof()` in your code? Do you typically use `sizeof()` with a variable name or a data type? While at first glance the distinction may not seem to matter that much, at a deeper level it matters a lot. Consider the following example.

### **Using `sizeof()`, a bad example**

```
int nVar;
...
DumpHex( &nVar, sizeof(int) );
```

DumpHex() is a general purpose routine that will dump out an arbitrary byte range of memory. The first argument is a pointer to a block of memory and the second argument is the number of bytes to dump.

Can you spot a possible problem in this example? The sizeof() in this example is operating on the int data type and not on the variable nVar. What if nVar needs to be changed in the future to a long data type? Well, sizeof(int) would have to be changed to sizeof(long). A better way to use sizeof() is as follows.

```
Using sizeof(), a good example  
int nVar;  
...  
DumpHex( &nVar, sizeof(nVar) );
```

In this new example, sizeof() now operates on nVar. This allows DumpHex() to work correctly no matter what the data type of nVar is. If the type of nVar changes, we will not have to hunt down in the code where the old data type was explicitly used.

## 7.15 Avoid Deeply Nested Blocks

There are times when, for any number of reasons, you end up writing code that is deeply nested. Consider the following example.

```
Deeply nested code  
void DeepNestFunction( void )  
{  
    if (test1) {  
        (more code)  
        if (test2) {  
            (more code)  
            if (test3) {  
                (more code)  
                if (test4) {  
                    (more code)  
                }  
            }  
        }  
    }  
}
```

```

    }
    }
}

} /* DeepNestFunction */

```

While this nesting is only four deep, I've had times when it would have gone ten deep. When nesting gets too deep, the code becomes harder to read and understand. There are two basic solutions to this problem.

Unroll the tests. The first solution is to create a boolean variable that maintains the current success or failure status and to constantly retest it as follows.

```

Unrolling the deep nesting
void UnrollingDeepNesting( void )
{
    BOOL bVar=(test1);
    if (bVar) {
        (more code)
        bVar = (test2);
    }
    if (bVar) {
        (more code)
        bVar = (test3);
    }
    ...
} /* UnrollingDeepNesting */

```

Call another function. The second solution is to package the innermost tests into another function and to call that function instead of performing the tests directly.

## 7.16 Keep Functions Small

The primary reason to keep functions small is that it helps you manage and understand a programming problem better. If you have to go back to the code a year later to modify it, you may have forgotten the small details and

have to relearn how the code works. It sure helps if functions are small.

As a general rule, try to keep functions manageable by restricting their length to one page. Most of the time functions are smaller than a page and sometimes they are a page or two. Having a function that spans five pages is unacceptable.

As a general rule, try to keep functions under one page.
--

If a function starts to get too large, step back a moment and try to break the function into smaller functions. The functions should make sense on their own. Remember to treat a function as a method that transitions an object from one valid state to another valid state. Try to come up with well-defined, discrete actions and write functions that perform these actions.

## **7.17 Releasing Debugging Code in the Product**

Is there such a thing as a debug build and a retail build of your product? Should there be? No, I do not think so! Let me explain why. I believe that any released application should be able to be thrown into debug mode on the fly at any time.

In the applications that I develop, I have a global boolean variable called `bDebugging` that is either `FALSE` or `TRUE`. I place what I consider to be true debugging code within an if statement that checks `bDebugging`. This is usually done for debugging code that adds a lot of execution overhead. For debug code that does not add much overhead, I just include the code and do not bother with `bDebugging`.

The benefit of doing this is that there is only one build of your product. That way, if a customer is running into a severe problem with your product, you can instruct the customer how to run your product in debug mode and quite possibly find the problem quickly.

I do not consider `WinAssert()` and `VERIFY()` to be debugging code. In the programs that I write, `WinAssert()` and `VERIFY()` are not switchable by `bDebugging`. Instead, they are always on. The reasoning is simple. Would

you like to know a bug's filename and line number in your program or would you just like to know that your program crashed somewhere, location unknown?

WinAssert() and VERIFY() are not debugging code.

If you object to the users of your product seeing assertion failures and run-time object verification failures, I recommend that you instead silently record the error in a log file. By doing this, you will have some record of what went wrong in the program when the customer calls you.

In a program that ships with WinAssert() and VERIFY() on, the program alerts the user to the exact filename and line number of a problem. If the fault-tolerant syntax is used, the program continues to run. Oftentimes, just knowing that a program failed at this one spot is enough to scrutinize that section of the code and find the problem.

It is important that the fault-tolerant forms of WinAssert() and VERIFY() be used. Doing so ensures that the program continues to run after a fault.

The fault-tolerant forms of WinAssert() and VERIFY() should always be used.

Sometimes a filename and line number are not enough to track down a problem. At times like these, a stack trace is highly desirable.

## 7.18 Stack Trace Support

A key debugging tool that I use for tracking down problems in my code is utilizing stack trace dumps from a fault. Sometimes only knowing the filename and line number of a fault is not enough to track down a problem. In these cases, the call history that led up to the problem is often enough.

For example, I once had a program that was faulting at a specific filename and line number. This code was examined thoroughly but no problem could be found. So, a stack trace of the fault was obtained from the customer, which assisted me in pinpointing the problem immediately. As it turned out,



a newly added feature had caused a reentrancy problem to occur in old code.

Most development environments today provide sophisticated tools that allow the developer to quickly pinpoint problems in their code. What do you do when a customer calls up with a fault that you cannot reproduce? The customer is certainly not running the development environment that you are running.

My solution to this problem is to add full stack trace capabilities into the application itself. At every point in the stack trace, a filename and line number are obtained.

I build stack trace support into my applications.

Unfortunately, the solution is specific to the underlying environment, so I cannot give a general solution, but I will do my best to describe the technique that I use.



### **7.18.1 Implementing Stack Trace Support**

*Obtaining filename and line number information.* The most important piece of information to which access is needed is debugging filename and line number information. Under a Microsoft development environment, obtaining this information is done in two steps. The first step is to tell the compiler to generate line number information. Under Microsoft C8, this is done with the /Zd command line option which results in the .obj files containing the line number information. The second step is instructing the Microsoft segmented executable linker to produce a .map file. The /map command line option is used.

*Translate the filename and line number information.* The .map file contains the filename and line number information in a human readable form. A program needs to be written that takes this .map text information and translates it into a form that is easily readable by the stack trace code.

*Walking the Stack.* This is the toughest part because it is so specific to the

platform that you are using. If walking the stack is not provided as a service by the operating system, you may want to consider walking the stack yourself. This is what I do. Luckily, Windows now provides stack walking support through the TOOLHELP.DLL system library. For Windows, the functions of interest are StackTraceFirst(), StackTraceCSIPFirst() and StackTraceNext().

*Mapping addresses to filename and line numbers.* As you walk the stack, the only piece of information available to you is an address. Somehow you need to map this back to the information you stored in the binary file representation of the .map file. Again, this is specific to the environment you are working on. Under 16-bit protected-mode Windows, far addresses are really composed of a selector and offset. The trick is to map the selector back to a segment number because the segment number is what is specified in the .map file. This is done in two steps. The first step is to map the selector to a global memory handle by using GlobalHandle(). The second step is then to map this global memory handle to a segment number by calling GlobalEntryHandle(). Both functions are provided by TOOLHELP.DLL. You can now look up the filename and line number.

You now have superior stack trace support built right into your application. It is superior because the stack trace gives filename and line numbers instead of the hex offsets usually given by system level stack traces.



### **7.18.2 Enhancements**

If you implement stack trace support in your application, I have some enhancements to suggest to you. I would highly recommend that you first get the basic stack trace support working before tackling these enhancements.

*Hooking CPU faults.* In protected memory environments, if your program accesses memory that does not belong to it, the program faults and the operating system halts the program. If possible, try to hook into this fault and produce a stack trace! For Windows, TOOLHELP.DLL provides the InterruptRegister() and InterruptUnRegister() functions that allow programs to hook their own faults. This requires some assembly language

programming.

*Hooking parameter errors.* Under Windows, the kernel is performing error checks on the parameters being passed to Windows API calls. It is possible to hook into the bad parameter notification chain. This is done by using TOOLHELP.DLL, which provides NotifyRegister() and NotifyUnRegister().

*Displaying function arguments.* As you walk the stack, try to parse what arguments were passed to the function along with the filename and line number. This is tricky but it is doable and well worth the effort. Most faults that cause stack traces are caused by an invalid argument in some function call. Spotting this in the stack trace then becomes easy.

### **7.18.3 The Benefits**

I have implemented full stack trace support along with hooking CPU faults, hooking Windows kernel parameter errors and displaying function arguments. What are the benefits? Great customer relations! In most cases, a stack trace is enough to track down a problem. In other words, I can track down a problem without first having to reproduce the problem. Customers begin to trust that a reported problem will get fixed and you end up with a robust product that the customer begins to trust and rely upon.

## **7.19 Functions Have a Single Point of Exit**

This has more to do with writing functions that are easily maintainable than anything else. If a function has one entry point, a flow of control and one exit point, the function is easier to understand than a function with multiple exit points.

It also helps eliminate buggy code because using a return in the middle of a function implies an algorithm that does not have a straightforward flow of control. The algorithm should be redesigned so that there is only one exit point.

In a sense, a return in the middle of a function is just like using a goto

statement. Instead of transferring control back to the caller at the end of the function, control is being passed back from the middle of the function.

## 7.20 Do Not Use the Goto Statement

I agree with the majority opinion that goto statements should be avoided. Functions with goto statements are hard to maintain.

## 7.21 Write Bulletproof Functions

Who is responsible for making sure that a function gets called and used properly? Is it up to the programmer? Or is it up to the function that gets called? Let's face it, programmers make mistakes. So anything that can be done on the part of the function to ensure that it is being used properly aids the programmer in finding problems in the program.

Consider a `GetTextOfMonth()` function. It takes as an argument a month, zero through eleven inclusive, and returns a long pointer to a three-character string description of the month. A naturally simple solution is as follows.

```
GetTextOfMonth() function, no error checking
LPSTR APIENTRY GetTextOfMonth( int nMonth )
{
    CSCHAR TextOfMonths[12][4] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    return (TextOfMonths[nMonth]);
} /* GetTextOfMonth */
```

The only problem with this code is what happens when the input `nMonth` is not in the proper range of zero to eleven? The returned pointer points to something, but if treated as a string, it is more than likely much longer than a three-character string. If this string is being used in a `printf()` statement, the resultant buffer has a high likelihood of being overwritten, trashing

memory beyond the end of the buffer and causing even more problems that need to be tracked down.

The solution is to make `GetTextOfMonth()` completely bulletproof so that any value passed into it returns a pointer to a three-character string. One possible solution is as follows.

```
GetTextOfMonth() function, with error checking
LPSTR APIENTRY GetTextOfMonth( int nMonth )
{
    CSCHAR szBADMONTH[]="???";
    LPSTR lpMonth=szBADMONTH;
    WinAssert((nMonth>=0) && (nMonth<12)) {
        CSCHAR TextOfMonths[12][4] = {
            "Jan", "Feb", "Mar", "Apr", "May", "Jun",
            "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
        };
        lpMonth = TextOfMonths[nMonth];
    }
    return (lpMonth);
} /* GetTextOfMonth */
```

Notice how a fault-tolerant form of `WinAssert()` is being used. This ensures that a full stack trace is logged if the input parameter is invalid.

Should this fault-tolerant code ever be removed? Once you get the code working that uses `GetTextOfMonth()`, you know there are no bugs, right? No, I do not think so! Do you know that your entire program is bug-free? There could be a bug in some totally unrelated part of the program that is causing a memory overwrite. If it just happens to overwrite a month number that you have stored in memory, you are in big trouble once again. Or what happens when you go back a year latter to modify the code that uses `GetTextOfMonth()`? You may introduce a subtle bug.

The best way to write a bug-free program is to keep all the defenses up at all times. At least this way, you will know when there is a problem in your program.

You may not know what the problem is, but just knowing that there is a bug is important information for maintaining a quality product.

The best way to write a bug-free program is to keep all the defenses up at all times.

## 7.22 Writing Portable Code

C is so successful because it is so flexible, flexible, that is, to the compiler writer because many key issues are left to the compiler writer to specify how they should work. This was done so that each implementation of C could take advantage of how a particular machine architecture works.

For example, what is the sign of the remainder upon integer division? How many bytes are there in an int or long or short? Are members of a structure padded to an alignment boundary? Does a zero-length file actually exist? What is the ordering of bytes within an int, long or short?

Most compilers provide a chapter or two in their documentation on how they have implemented these and many more implementation-defined behaviors.

If writing portable code is important to you, I would suggest that you thoroughly read these chapters and adopt programming methodologies that avoid implementation dependent behavior.

## 7.23 Memory Models

Due to the segmented architecture of the Intel CPU, compiler vendors provide the option of creating a program in one of four basic memory models.

*The small memory model.* This model allows for less than 64K of data and less than 64K of code. This model is great for quick and dirty utility programs. It also produces the fastest program since there is never any need to reload a segment register.

*The compact memory model.* This model allows for more than 64K of data and less than 64K of code. It is ideal for small programs that crunch through a lot of data. The program is still fast, but there is a slight speed penalty for accessing far data.

*The medium memory model.* This memory model allows for less than 64K of data and more than 64K of code. This is the memory model that Microsoft recommends using for programming in Windows. It allows for lots of code and a small amount of data. By using mixed-model programming, you can gain access to more than 64K of data.

*The large memory model.* This allows for more than 64K of data and more than 64K of code. It is the memory model that most closely matches flat memory model architectures.

These memory models are complicated by the fact that there is something called mixed-model programming. The four basic memory models essentially provide default code and data pointer attributes. A code or data pointer is either a 16-bit near pointer or a 32-bit segment/offset pointer. These near and far attributes are only defaults. Mixed-model programming allows the near/far attributes to be specified on a pointer-by-pointer basis.

My advice to you is to use the large memory model, unless you are ultra concerned with speed. The industry is moving away from segmented architectures toward flat memory model programming, where there are no segments.

Use the large memory model. This will aid porting to flat memory model architectures.
---

By using the large memory model now, you ease the eventual porting of your software to the flat memory model.

## 7.24 Automated Testing Procedures

An automated testing procedure is a function that is designed to

automatically test your code for you -- code that you think is already bug-free. A key part of most testing procedures is their use of random number generator class objects.

Let's suppose that you have just implemented a B-Tree disk-based storage heap for fast access to your database. How are you going to really test it? You could code examples that use the B-Tree class in order to test edge conditions. This is a good idea anyway, but what do you code next?

A solution that I have found useful and highly effective is to use a random number generator to create data sets that are then thrown at the module to be tested. A random number generator is useful because if a problem is discovered, it can be absolutely recreated by using the same random number seed.

A random number generator is an important part of an automated testing procedure.
---

In the case of the B-Tree code, you could randomly decide to add or delete a record from the tree and you could randomly vary the size of the records being added. You could also decide to randomly restart the test. As you add records into the database and read them back, how do you verify the integrity of the randomly sized record? One slick technique is to use another random number generator to fill in the record with random values. The slick part is that all you need to save around in memory to validate the record when it is read back in is the random number seed that was used to generate the random record, not the entire record itself.

Another big advantage of using random number generators is that given enough time, they can test virtually all cases and code paths. It is a lot like throwing darts. If you keep on throwing a dart at the dart board, the center target is eventually hit. It is only a matter of time. The question is not if the target is hit, but when.


What you are doing with the automated testing procedure is taking a module that is considered to be bug-free and subjecting it to a torture test of



random events over time. Assuming there is a bug in the module, the automated testing procedure will find it eventually.

If there is a bug in a module, an automated testing procedure will eventually find it.

It is important that the automated testing procedure be written so that it is capable of generating all types of conditions and not just the normal set of conditions. You want to make sure that all the code in the module gets tested.

 Do automated testing procedures actually work? Yes! They are what turned up the MS-DOS lost cluster bug and the Windows task-switching bug described in [§3.1](#). I was putting my own code through a rigorous automated testing procedure and every once in a while the underlying kernel would fail. I guess the use of probability theory pays off.

## 7.25 Documentation Tools

Programmers hate to write documentation, but they love to write code and most programmers are willing to comment their code to some degree. An even bigger problem is that even if documentation does exist, it is more than likely out of date because it hasn't been maintained to reflect code changes.

My solution to this problem is to accept the fact that external documentation is not going to be produced directly. Instead, I am going to produce it indirectly.

By having all programmers follow a common documentation style in the entire project, it is possible to write a program that scans all source files and produces documentation.

I use markers in comment blocks to assist me in parsing my comments. For example, module comment blocks begin with `/*pm, APIENTRY` function

comment blocks begin with `/*pf` and LOCAL function comment blocks begin with `/*p`.

In practice, this works great. The AUTODOC program that I use scans all sources files and produces a Microsoft Quick Help file as output. The Brief editor that I use supports Quick Help. I now have instant access to all APIENTRY function documentation at the touch of a key.

## **7.26 Source-Code Control Systems**

If you are not already using a source-code control system, I would highly recommend that you get one. I like them because they give me access to the source as it existed all the way back to day one. It is also essential for tracking down problems in released software. You may end up with two or three different versions of your software that are all in active use. The source-code control system gives you easy access to the source of any particular version of your software.

Most source-code control systems follow a get and put methodology. Getting a source file gives the "getter" editing privileges to the source. When changes are complete, the source is put back.

Before I put back any source, I produce a difference file and review all the changes that I have made to the source. On more than one occasion this has saved me from including a silly programming bug.

Always review changes before checking source code back in.
--

### **7.26.1 Revision Histories**

An important part of maintaining software is keeping an accurate log of what changes were made to a module and why. Rather than keeping this information in the source file itself, I prefer to use the source-code control system.

In the source-code control system that I use, a put will prompt me to enter a description of the changes that I have made to the source file.

The entire revision history is available at any time and is maintained by the source-code control system. In modules that get changed a lot, this technique keeps around the full revision history without cluttering up the source file.

## 7.27 Monochrome Screen

### 7.27.1 The Windows Developer

A monochrome monitor is a must for the Windows-based developer. You can configure the debugging kernel to send debug messages to either a serial communications port or the monochrome monitor. This is configured in the DBWIN.EXE program, which is provided with Microsoft C8. A monochrome monitor is preferred because it is a lot faster when you get a lot of debug messages at once.

In addition to the system generating debug messages, the programmer can generate them as well by calling `OutputDebugString()`. The prototype for it is as follows.

```
OutputDebugString() prototype in windows.h (v3.1)  
void WINAPI OutputDebugString(LPCSTR);
```

`OutputDebugString()` should not be called in the final release of your software. You do not want messages going to your customer's communication port. In my code, I control this by calling `OutputDebugString()` only if I detect that the debugging kernel is running. To detect if you are running under debug Windows, use the `GetSystemMetrics(SM_DEBUG)` call. It returns zero under retail Windows and a non-zero value under debug Windows.

Another benefit of using a monochrome screen is that most Windows debugging tools have an option to run on the monochrome screen. This way you can see the debug screen and your main screen at the same time.



## 7.27.2 The MS-DOS Developer

Just as in Windows, most MS-DOS debugging tools have an option to run on the monochrome screen. What do you do if you want to send messages to the monochrome screen?

An `OutputDebugString()` that can be used by MS-DOS programmers is as follows.

### **OutputDebugString() for MS-DOS programmers**

```
void APIENTRY OutputDebugString( LPSTR lpS )
{
    LPSTR lpScreen=(LPSTR)0xB0000000; /* base of mono screen
    */
    int nPos=0;                        /* for walking lpS
string */
    /*--- Scroll monochrome screen up one line ---*/
    _fmemcpy( lpScreen, lpScreen+2*80, 2*80*24 );
    /*--- Place new line down in 25'th line ---*/
    for (int loop=0; loop<80; ++loop) {
        lpScreen[2*(80*24+loop)] = (lpS[nPos]?lpS[nPos++]:'
');
    }
}

/* OutputDebugString */
```

The monochrome screen is memory mapped and is located at segment 0xB000. Every character on a monochrome screen is actually composed of 2 display bytes. One byte is the character to display and the other byte contains attribute information such as blinking, inverted, and so on.

This code works by first scrolling the monochrome screen by performing a memory copy. Next, the string is placed into line 25 of the monochrome screen. The string is placed space padded at the end to make sure that the previous contents of the twenty-fifth line are overwritten.

## 7.28 Techniques for Debugging Timing Sensitive Code

Application code should never have any timing dependencies. However, system level or interrupt code will more than likely have timing constraints. An example is an interrupt handler for a synchronous communications protocol. These drivers can be especially hard to debug because there is always communications traffic on the line and the protocol itself is timing sensitive. Using `OutputDebugString()` to help you debug the code wastes too much time and affects the timing sensitive code you want to debug, so an alternative is needed.

### 7.28.1 `PutMonoChar()` Function for MS-DOS

One technique that I have used successfully to debug timing sensitive code is to write a few informative characters directly into the monochrome screen video memory, in effect displaying a message on the monochrome monitor. For example, `PutMonoChar()` places a character at a specific row and column on the monochrome screen.

#### **`PutMonoChar()`, for MS-DOS**

```
void APIENTRY PutMonoChar( int nRow, int nCol, char c )
{
    if ((nRow>=0) && (nRow<25) && (nCol>=0) && (nCol<80)) {
        *(LPSTR) (0xB0000000+2*(nRow*80+nCol)) = c;
    }
}

/* PutMonoChar */
```

`PutMonoChar()` works by first validating that the input `nRow` and `nCol` are valid. It then writes the character directly into monochrome screen video memory.

The advantage of using `PutMonoChar()` as opposed to `OutputDebugString()` for debug messages is that it is so much faster and is unlikely to adversely affect the timing sensitive code you want to debug. This is because `PutMonoChar()` is just placing one character down instead

of `OutputDebugString()`, which is placing an entire line down and scrolling the entire monochrome screen.

---

Copyright © 1993-1995, 2002-2003 Jerry Jongerius  
This book was previously published by Person Education, Inc.,  
formerly known as Prentice Hall. ISBN: 0-13-183898-9

# Preface [Writing Bug-Free C Code](#)

[Quick Overview of the Book](#)  
[How It All Started](#)

[Contacting the Author](#)  
[Acknowledgments](#)

**Note to this online book:** On April 29, 2002, I reacquired the publishing rights to my book (from Prentice Hall; published in January 1995), and have decided to publish it online, where it is now freely available for anyone to read. The book is showing its age, but for people who still program in C, the techniques described in this book are still a 'little gem' worth knowing about. Enjoy! - *Jerry Jongerius*, [jerryj@duckware.com](mailto:jerryj@duckware.com)

This book describes an alternate class methodology that provides complete data hiding and fault-tolerant run-time type checking of objects in C programs. With it, you will produce code that contains fewer bugs.

The class methodology helps to prevent bugs by making it easier to write C code. It does this by eliminating data structures (class declarations) from include files, which makes a project easier to understand (because there is not as much global information), which makes it easier to write C code, which helps to eliminate bugs. This class methodology, which uses private class declarations, is different from C++, which uses public class declarations.

The class methodology helps detect bugs by providing for both compile-time and run-time type checking of pointers (handles) to class objects. This run-time type checking catches a lot of bugs for you since invalid object handles (the cause of a lot of bugs) are automatically detected and reported.

We have all, at some point in our programming careers, spent several hours or days tracking down a particularly obscure bug in our code. Have you ever stepped back and wondered how following a different programming methodology might have prevented such a bug from occurring or have automatically detected it? Or have you tracked down the same type of bug several times?

The key to eliminating bugs from your code is learning from your mistakes.

I have had my fair share of bugs, but when one does occur, I immediately try to institute a new programming methodology that helps prevent and/or automatically detect the bug. That is what this book is all about. It documents the techniques that I use to write virtually bug-free code. You may not agree with 100 percent of the techniques I use and that is OK and only fair, since every programmer has his or her own style. Where you do disagree, however, I challenge and urge you to come up with an alternate methodology that works for you.

## Quick Overview of the Book

The examples in this book were taken directly from a large production Windows application. Because of this, some of the names you will find are tied to the environment in which the program resides. However, I prefer to show you code taken directly from a real-life production application rather than code that has been made up simply because this book is being written.

["Understand Why Bugs Exist."](#) Chapter 1 explores why I think bugs exist in programs today. This is the first step in writing bug-free code. How can we hope to eliminate bugs from our code unless we understand how they come to exist?

["Know Your Environment."](#) In order to implement your own programming methodologies that detect bugs, it is important to fully understand your programming environment. Chapter 2 examines the C language and preprocessor in an attempt to show you something new about your programming environment, even if you have been programming in C for many years. For example, did you know that `buffer[nIndex]` is equivalent to `nIndex[buffer]`? Chapter 2 also contains a few programming puzzles to get you thinking about your programming environment.

["Rock-Solid Base."](#) Chapter 3 stresses the importance of coding upon a rock-solid base. Without a bug-free base to code upon, how can you be expected to write bug-free code? Chapter 3 suggests what should be done.



["The Class Methodology."](#) The class methodology is the key to writing bug-free code and is described in Chapter 4. The class methodology allows complete data hiding and fault-tolerant run-time type checking of objects in C. It accomplishes complete data hiding by moving data declarations out of include files and into the one source file that needs the declaration (very much unlike the public class declarations in C++). The breakthrough to this class methodology is that pointers (handles) to objects of a class can be type checked by the compiler in other source files that use the class but do not have access to the data declaration.

["A New Heap Manager."](#) The heap manager that comes with C does not detect programmer bugs. Chapter 5 shows how to create a heap manager that is rock-solid. In addition, the heap manager is needed to support the class methodology.

["Designing Modules."](#) Now that everything is in place for writing bug-free code, how should a module be designed? Chapter 6 shows that the key to designing and coding modules is to always code what to do, not how to do it.

["General Tips."](#) Chapter 7 contains a variety of tips for writing bug-free code. For example, a key to writing bug-free code is learning from your mistakes. Also, designing stack trace support into an application allows you to track down problems without having to reproduce the problem.

["Style Guide."](#) Chapter 8 describes the style that I use to write C code. While every programmer has his or her own way of coding, I feel it is important that a style guide exist in written form.

["Conclusion."](#) Chapter 9 contains concluding remarks. The class methodology and run-time type checking are key features presented in this book.

["Code Listings."](#) This appendix brings together all the code presented in the book into one convenient location.

## How It All Started

It all started when I was a sophomore in high school. Back then, I was introduced to an Apple II computer and was completely amazed by what the computer could do. The Apple had great graphics with plenty of games and educational software. I began to wonder how all the magic was accomplished.

It wasn't long before I wrote my first BASIC program. Shortly thereafter, however, I encountered my first bug. The only tools and resources available were the computer, the computer manuals and myself. Those were trying times. It taught me to think things through before jumping into coding.

In my probing around into how the computer worked, I quickly came across something called assembly language and the command CALL -151 (the equivalent of DEBUG for the PC, but in the Apple II ROM). The speed at which programs executed in relation to BASIC was remarkable. At that time, all my assembly language programming was done by hand. The assembly program was entered by typing in the hex digits of the opcode and operands. To this day I still remember a lot of opcodes even though I have not coded in 6502 assembly for many years. I quickly realized the benefits of designing a piece of code before writing it because making any changes essentially meant rewriting the entire assembly program. Later, I purchased an assembler.

My quest for finding out how the computer worked has never ended. I disassembled (reverse engineered) and commented the entire Apple II Disk Operating System (DOS 3.3 at that time), the bootstrap ROMs and parts of the BASIC interpreter ROMs. I started writing programs that performed disk protection, added keyboard buffers, and so on -- in other words, systems programming.

Today, I am working on an 80486 66-MHz DX2 with 32 megabytes of memory, local bus video and 600 megabytes of hard disk space -- quite a change from a 1-MHz, 64 kilobytes of memory, 140-kilobyte floppy disk drive Apple II. The hardware has changed drastically, but the methodologies used to write programs haven't changed as fast. The quality

of code certainly hasn't improved by several orders of magnitude.

After college, I started working at Datametrics Systems Corporation, a firm specializing in the performance of Unisys mainframes. Their goal was to produce a top-of-the-line performance monitoring package that ran under Microsoft Windows. A year later, that goal was realized with the release of ViewPoint 1.0. ViewPoint as it stands today is now approximately a quarter million lines of code and very stable.

ViewPoint originally started out under Windows 1.0. Shortly thereafter, Windows 2.0 was released. Since then, Windows 3.0 and now Windows 3.1 have become mega hits in the PC industry.

The great thing about Windows 3.0 was that programming bugs could be caught much quicker than before because programming errors caused a general protection fault instead of just trashing memory and more than likely hanging the PC. This is because under pre-3.0 Windows, programs had full access to all the memory in the PC, even memory of other processes. So a bug could trash anything and usually would. Windows 3.0 supported protected address spaces. Therefore, an access to a random memory location most likely would be detected (the infamous UAE, Unrecoverable Application Error) by the operating system and your program halted.

When I converted ViewPoint to Windows 3.0, it ran the first time. The protected-mode architecture turned up no programming bugs, not a single one! In fact, the ViewPoint 1.0 binary, which was targeted to Windows 2.0, a non-protected OS, can still be run under Windows 3.1, a protected OS, without any problems. This indicates that the program contains no invalid memory references and indicated to me that the programming methodologies that were used to write ViewPoint are valid.

In the last few years, the programming industry has moved from procedural programming to object-oriented programming (OOP). However, OOP alone does not solve the underlying quality control problems facing the industry. OOP alone does not make you a better programmer.

I wrote this book to document the programming methodologies that I have used to write ViewPoint under Windows. The methodologies were designed and developed using Microsoft C5 through C8. Since the introduction of Microsoft C/C++ 7.0, the code has been ported to compile under C++.

## **Contacting the Author**

If you have any comments on this book, I would like to hear them. I can be reached through the Internet at [jerryj@duckware.com](mailto:jerryj@duckware.com).

## **Acknowledgments**

Special thanks to P.J. Plauger for taking the time to listen to a first-time book author. Your help is greatly appreciated and invaluable.

I would like to thank everyone at Prentice Hall who helped to make this book a reality, especially Paul W. Becker and Patti Guerrieri.

Thanks to John Kelly, the president of Datametrics Systems Corporation, for allowing me to write a book that discloses the programming techniques that I developed at Datametrics Systems Corporation.

Thanks to Jim Kelliher, who provided feedback throughout the entire book writing process. Thanks to John McGowan, who provided feedback on the class methodology. Thanks to John Dripps, who helped during the early stages.

# Chapter 1: Understand Why Bugs Exist [Writing Bug-Free C Code](#)

[1.1 Small versus Large Projects](#)

[1.2 Too Many Data Structures in Include Files](#)

[1.3 Using Techniques That Do Not Scale](#)

[1.4 Too Much Global Information](#)

[1.5 Relying on Debuggers](#)

[1.6 Fixing the Symptom and Not the Problem](#)

[1.7 Unmaintainable Code](#)

[1.8 Not Using the Windows Debug Kernel](#)

[1.9 Chapter Summary](#)

Why do bugs exist and where in the development cycle do they creep in? Spending time and effort on the problem of understanding why bugs exist is the first step to writing bug-free code. The second step is to take action and institute policies that eliminate the problem or help detect the problem. Most important, make sure the entire programming staff knows about and understands the new policies.

The first step in writing bug-free code is to understand why bugs exist. The second step is to take action.

A good friend of mine who works at a different company started to use run-time parameter validation in the modules that he wrote and the code that he modified. Run-time parameter validation is a good idea. However, management and other programmers at the site were reluctant to make this programming methodology mandatory. Well, one day my friend was modifying some existing code on the project and while he was at it, he added parameter validation to the functions that he had modified. He tested the code and checked it back into the source-code control system. A few weeks later the code started to display parameter errors from code that it had called that had been written years ago. Unbelievably, some programmers wanted the parameter validation removed. After all, they reasoned, code that once worked was now producing errors, so it must be the new parameter validation code, not the old existing code.

This is an extreme example, but it demonstrates that everyone involved with a project must fully understand new programming methodologies instituted in the middle of a programming project.

## 1.1 Small versus Large Projects

What if you needed to write a hex dump utility to be called DUMP? The program takes as an argument on the command line the name of the file you wish to display in hex. Would it be written without a single bug? Yes, probably so, but why? Because the task is small, well defined and isolated.

So what happens when you are asked to work on project ALPHA, a contract programming project your company is working on for another company? The project is several hundred thousand lines long and has ten programmers already working on the project. The deadlines are approaching and the company needs your programming talent. Do you think you could jump right in and write new code without introducing bugs into the project? I couldn't, at least not without the proper programming methodologies in place to catch the programming errors any beginner on the project is bound to make.

Think of the largest project you have worked on. How many include files were there and what did the include files contain? How many source files were there and what did they contain? You had no problem working on the DUMP utility, so what makes the large project so difficult to work on? Why is the large project not simply like working on ten or one hundred small projects?

Programming methodologies must make it easy for new programmers to jump into a project without introducing a slew of new bugs.
--

Let's examine your small programming project. It consists of a couple of source files and a couple of include files. The include files contain function prototypes, data structure declarations, #define's, typedef's and whatever else. You have knowledge of everything, but because the number of files is relatively small, you can handle it all. Now, multiply this by ten or one hundred times for the large project and all of a sudden the project becomes unmanageable.

You have too much information in the include files to manage and now the

project is getting behind, so you add more people to the project to get it completed faster. This only compounds the problem, because now you have even more people adding information to the pool of information that everyone else needs to learn and know about. It is a vicious cycle that all too many projects fall into.

## 1.2 Too Many Data Structures in Include Files

We can all agree that one major problem in large projects is that there is too much information to become familiar with in a short period of time. If you could somehow eliminate some of the information, this would make it easier, since there would be less information to become familiar with.

The root of the problem is that there is too much information placed into include files, the biggest contributor being data structure declarations. When you start a project, you place a couple of declarations in the include file. As the project continues, you place more and more declarations in the include file, some of which refer to or contain previous declarations. Before you know it, you have a real mess on your hands. The majority of your source files have knowledge of the data structures and directly reference elements from the structures.

A technique that helps eliminate data structures from include files needs to be found.
--

Making changes in an environment where many data structures directly refer to other data structures becomes, at best, a headache. Consider what happens when you change a data structure. This change forces you to recompile every source file that directly, or more importantly indirectly, refers to the changed data structure. This happens when a source file refers to a data structure that refers to a data structure that refers to the changed data structure. A change to the data structure may force you to modify some code, possibly in multiple source files.

The [class methodology\\_§4](#) solves this data structure problem.

## 1.3 Using Techniques That Do Not Scale

As you have just seen, large projects have problems all their own. Because of this, you must be careful in selecting programming methodologies that work well in small projects as well as in large programming projects. In other words, a programming methodology must scale or work with any project size.

Programming methodologies must work equally well for both small and large projects.
---

Let's say that Joe Programmer institutes a policy that all data declarations must be declared in include files so that all source files have direct access to the data structures. He reasons that by doing this, he will gain a speed advantage over his competition and his product will be superior.

This may work for the first release of his product, but what happens when the size of the project grows to the point that it becomes unmanageable because there are too many public data declarations? His job is at stake. The programming methodology Joe chose worked great for the small project when it started, but it failed miserably when the project grew. And all successful small projects grow into large projects.

Make sure the programming methodologies that you develop work equally well for both small and large projects.

## **1.4 Too Much Global Information**

Global variables (variables known to more than one source file) should be avoided. Their usage does not scale well to large applications. Just think what eventually happens in a large application when the number of global variables gets larger and larger. Before you know it, there are so many of them that the variables become unmanageable.

Whenever you are about to use a global variable, ask yourself why you need direct access to the variable. Would a function call to the module where the variable is defined work just as well? Most of the time the answer is yes. If the global variable is modified (read and written), then you should



be asking yourself how the global variable is being acted upon and if it be better to specify what to do through a function call, leaving the how to the function.

Variables known to more than one source file should be avoided.

Some global variables are OK, but my experience has been that only a handful are ever needed, no matter how large a project gets.

## 1.5 Relying on Debuggers

Debuggers are great tools, but they are no substitute for good programming methodologies that help eliminate bugs in the first place.

Debuggers certainly make finding bugs easier, but at what price? Does knowing that there is a powerful debugger there to help you when you get in trouble cause you to design and write code faster and test it sooner than you should have, causing a bug that forces you to use the debugger? A bug that would not be there had you spent the time in the design and coding phases of the project.

I personally feel that programmers whose first course of action against a bug is to use a debugger begin to rely more and more on a debugger over time. Programmers who use a debugger only after all other options have been exercised begin to rely less and less on a debugger over time.

Instead of a debugger, why not look at the code and do a quick code review, using available evidence from the program crash or problem? Often, the evidence plus looking at the code pinpoints the problem faster than the time it takes to start the program in the debugger.

Use a debugger only as a last resort. Having to resort to a debugger means your programming methodologies have failed.

By now you are probably wondering how often I use the debugger in my programming environment. Because of all the programming methodologies that I have in my bag of techniques, not very often.

Your goal should be to develop and use programming techniques that catch your bugs for you. Do not rely on a debugger to catch your bugs.

## 1.6 Fixing the Symptom and Not the Problem

Let's say you encounter a bug in your code. How do you go about fixing the problem? Before you answer, what is the problem? Is the problem the bug itself, or is it what caused the bug to occur?

All too often, the bug or symptom is being fixed and not the actual problem which caused the bug in the first place. A simple test is to think how many times you have encountered essentially the same bug. If never, then great, you are already fixing the problem and not the symptom. Otherwise, give some careful thought to what you are fixing. You probably want to come up with a new programming methodology that helps to fix the problem once and for all.

When fixing a bug, fix the underlying problem or cause of the bug and not just the bug itself.
--

A prime example is storage leaks. A storage leak is defined as a memory object that is allocated but never freed by your program. In most cases, the storage leak does not directly cause problems, but what if it happens on an object that gets allocated and (not) freed a lot? You run out of memory at some point.

Running out of memory is what caused you to start looking for the problem. You eventually find the missing line of code that should have deallocated the memory object and add the memory deallocation into the program. Bug fixed, right?

Wrong! The underlying problem of storage leaks going undetected is still in the program. In fact, had the heap manager told you where there was a storage leak in your program, you would not have wasted your time tracking down the storage leak. To fix the problem once and for all, the heap manager must tell you that there is a storage leak. This programming

methodology is discussed in [Chapter 5](#).

## 1.7 Unmaintainable Code

We all have had the experience of modifying code that someone else has written, either to add a new feature or to fix a problem in a module. I don't know about you, but for me it is typically not an enjoyable experience because the code is often so hard to understand that I end up spending the majority of my time just figuring out what is going on.

Strive to write code that is understandable by other programmers.

Good code runs. Great code runs and is also easily maintainable. In your code reviews, you should look for code that not only works, but also for code that is straightforward in how it works. What good is a coding technique if it cannot be understood? Consider the following.

**Hard to understand code?**  
`nY = (nTotal-1)/nX + 1;`

Is this code fragment hard for you to understand? If not, then you know the technique being used. If you do not know the technique, then it is sure frustrating to figure out. This code is one way to implement the ceil function on the `nTotal/nX` value.

Properly commenting your code is a good first step. However, keep in the back of your mind that someone else is reading your code and avoid obscure programming techniques unless they are fully commented and documented in the project.

## 1.8 Not Using the Windows Debug Kernel

I am primarily a Microsoft Windows developer and it astounds me the number of times I have run commercial Windows applications on my machine only to have them bomb because the Windows debug kernel is

reporting a programming error such as an invalid handle. The error is in such an obvious place that you know the developers did not use the debugging kernel of Windows during their development. Why would anyone develop an application and not use the debugging kernel of the underlying environment? It catches errors in your code automatically.

The Windows development environment allows for running either the retail kernel or the debugging kernel. The retail kernel is the kernel as it is shipped to the customer. The debugging kernel has extra error checking not present in the retail kernel. Error messages from the kernel may then be redirected to a secondary monochrome screen attached to the system. This redirection is an option in the DBWIN.EXE program provided with Microsoft C8. You should run with the debugging kernel all the time.

Use the debugging kernel of your development environment.

The Windows SDK provides D2N.BAT (debug to non-debug) and N2D.BAT (non-debug to debug) batch files in the BIN directory to switch between debug and non-debug Windows. It is easy to accidentally leave your version of Windows in the non-debug mode. It has happened to me a couple of times. To detect this situation, I finally printed a special symbol in my application's status line to signify that it is being run under debug Windows. I suggest that you do something similar in a place that is easy to spot in your Windows application. To detect if you are running under debug Windows, use the `GetSystemMetrics(SM_DEBUG)` call. It returns zero under retail Windows and a non-zero value under debug Windows.

## 1.9 Chapter Summary

- The first step in writing bug-free code is to understand why bugs exist. The second step is to take action. That is what this book is all about.
- Programming methodologies that are developed to prevent and detect bugs must work equally well for both small and large programming projects.

- A technique that helps eliminate data structure declarations from include files needs to be found. Doing so will allow programmers to come up to speed on an existing project much quicker.
- Global variables that are known to more than one source file should be avoided. Global variables make it hard to maintain a project.
- Debuggers should be used only as a last resort. Having to resort to a debugger means that your programming methodologies used to detect bugs have failed.
- When you fix a bug, make sure you are really fixing the underlying cause of the bug and not just the symptom of the bug. Ask yourself how many times you have fixed the same type of bug.
- Strive to write code that is straightforward and easily understandable by others. Avoid writing code that pulls a lot of tricks.
- Finally, make sure that you use the Windows debug kernel all the time. It contains extra error checking that can automatically detect certain types of bugs that go undetected in the retail release of Windows.

# Chapter 3: Rock-Solid Base [Writing Bug-Free C Code](#)

[3.1 System Functions Contain Bugs](#)

[3.2 Using Macros to Aid Porting](#)

[3.3 Using WinAssert\(\) Statements](#)

[3.4 Naming Conventions](#)

[3.5 Chapter Summary](#)

Would you build a skyscraper without a proper, solid foundation? Of course not. Would you build a large application without a rock-solid base system? I wouldn't, and yet I get the feeling that this is happening every day. Do you consider the standard C library to be a rock-solid base? Before you answer this, I need to clarify what I consider to be rock-solid.

A rock-solid function must first of all be bug-free itself. The function must provide a clean, intuitive interface. What hope would you ever have if you were constantly making mistakes in how a function is called? Function names must clearly state what the function does. What good is a function named `DoIt()`? The function must detect and report invalid function arguments. The function should be fault-tolerant. The program you are working on should not crash simply because you called a function incorrectly.

Before you can write bug-free code, you must have a bug-free, rock-solid base.

Do you now consider the standard C library to be a rock-solid base? Many functions are, but many functions are not. Consider the heap management routines in the C library, specifically, the `free()` function. The `free()` function deallocates a block of memory that was previously allocated through the `malloc()` function call.


What happens when you pass `free()` a completely random value, or a value that you have already previously passed to `free()`? Your program may bomb immediately. If it doesn't, the heap may be corrupted. If it isn't, some memory may have been overwritten. The point is that not all C library functions are rock-solid. Why not first code a layer on top of the system base that is rock-solid?

Not all C library functions are rock-solid. A layer that is rock-solid needs to be coded.

As you code your program, you need to consider the current program as it stands as the base for whatever new features you are putting in. Once done, this is the new base for the next set of features. As you code, make sure that the current base is rock-solid; that it is fault-tolerant and that it catches incorrect usage of functions. If the base is not rock-solid, you need to make it rock-solid.

### 3.1 System Functions Contain Bugs

Your underlying operating system or development environment has bugs in it. Since there is no such thing as a completely bug-free system, try to find out as much as you can about the environment you are working on. Try to obtain bug lists if they are available.

 Microsoft has recently started the Microsoft Developer Network. It is a program that is intended to get as much information and technical resources as possible into the hands of developers. The program distributes information in the form of a CD and a Windows-based browser. I highly recommend this program to developers. The Microsoft Developer Network can be reached through the Internet at <http://msdn.microsoft.com>

#### 3.1.1 A MS-DOS Bug

An example of a bug that was found in the MS-DOS operating system involves lost clusters. It was detected by an [automated testing procedure §7.24](#). As the automated testing function would run, invariably it would eventually run out of disk space, but the function never created any file large enough to even come close to running out of disk space. As it turned out, disk clusters were being lost by the operating system on a regular basis and eventually the disk would run out of free clusters. The only way to recover the lost clusters was to run the CHKDSK program provided by the operating system.

This bug is a confirmed bug in MS-DOS 3.2 and MS-DOS 3.3, the only versions available when the bug was discovered. The bug does not exist in MS-DOS 5.0 and later versions.

**C program that shows MS-DOS version 3.3 lost cluster bug**

```
#include <fcntl.h>
#include <sys\types.h>
#include <sys\stat.h>
#include <io.h>
#include <stdio.h>

int main(void)
{
    char c;
    int fh = creat( "it.tmp", S_IREAD|S_IWRITE );
    lseek( fh, 81920, SEEK_SET );
    write( fh, &c, 1 );
    chsize( fh, 81920 );
    lseek( fh, 122880, SEEK_SET );
    write( fh, &c, 1 );
    close( fh );
    return 0;
} /* main */
```

Running this program repeatedly under MS-DOS 3.2 or MS-DOS 3.3 will cause the lost cluster bug. Run CHKDSK to recover the lost clusters.



### **3.1.2 Windows Bugs**

As any early Windows developer will tell you, Windows was a buggy operating system. Over the course of five years, I accumulated over six three-ring binders full of correspondence with Microsoft support concerning bugs in the Windows system.

With the introduction of Windows 3.0, I noticed a significant drop in the number of problems that I was reporting. I attribute this reduction to the fact that Windows 3.0 was a protected-mode operating system (previously it was not). Many types of errors cause a CPU fault and the error can be



pinpointed immediately. However, the Windows 3.0 system was still a little shaky when it came to DOS boxes, the emulation of a MS-DOS PC in a Windows window.

With the introduction of Windows 3.1, the number of bugs that I report has dropped dramatically. I consider Windows 3.1 to be a stable operating environment. In fact, on my development machine, I stay in Windows all day and use DOS boxes to run my MS-DOS applications. My machine may crash once a week. Not too bad, but still room for improvement.

However, there is a particularly nasty bug I have found that still exists in Windows 3.1. Luckily, the likelihood that you will ever encounter the bug is almost nil. The bug is that sometimes the Windows multitasker does not switch to the correct PSP when a Windows application performs disk I/O. What this means is that a file handle that should be correct is not, because the program is in the open file context of another application!

A little background may be needed to explain why this can even occur. When Windows multitasks applications, it has to keep a tremendous amount of information around for each application. One piece of that information is the open files context. In order to significantly speed up task switching in Windows, the open files context is not switched until the task actually performs an operation that requires the open files context. Reading and writing a disk file is an example of an operation that would cause a true open files context switch. Since most applications rarely perform disk I/O relative to the amount of CPU time they need, this delayed context switch of the open files context speeds up multitasking.

The bug occurs when, in the course of multitasking applications, the Windows kernel incorrectly thinks that it is truly in the open files context of the currently running application, when, in fact, it is in the open files context of another application. Microsoft has been slow to fix this problem since the circumstances that cause the bug to occur are extreme.

The only reason I found this bug in the first place is because of an automated testing procedure I used to test the correctness of a new module. This procedure caused almost continuous I/O to occur and every once in a

great while, an I/O would fail under low memory conditions.

A Windows 3.1 bug. There is another bug that I found that anyone can reproduce easily on any Windows 3.1 machine. First, run the standard Notepad application. Go into the file open dialog and click with the mouse on the OK button. Now press and hold down the space bar. While continuing to hold down the space bar, press the ESC key. This causes a General Protection Fault. The problem has to do with the dialog manager inside the Windows kernel. The fault occurs because both an IDOK and an IDCANCEL are being sent to the dialog callback procedure when in reality, only one message should be sent. This is a problem with all dialogs in all applications. However, the application may or may not fault. It just depends upon how the application was written to respond to the IDOK and IDCANCEL messages. This bug is fixed in Windows 3.11.

### 3.1.3 What to Do

The point in demonstrating to you that bugs do exist in MS-DOS and Windows is to emphasize that sometimes even system level functions fail. Code that you think never fails is bound to fail sometime.

My reaction to having system level functions fail me has been to provide another layer of code between my application code and the system level functions that checks for assumptions that I am making. At some point, you write code that makes an assumption. Consider the following code.

```
Code with an assumption  
int fh = open(...);  
if (fh!=-1) {  
    close(fh);  
}
```

Do you see what the assumption is? The assumption being made in this code is that the close() function never fails. Well then, why not assert this? The close() returns zero for success, otherwise non-zero for failure.



Provide a code wrapper around all system calls.

The best solution is to provide a wrapper function around each and every system call. Assert any assumptions that are being made within this wrapper function. Placing the assert in the wrapper function once instead of every place it is being called is a lot less error prone.

## 3.2 Using Macros to Aid Porting

With all the different machine architectures that are in use today, how in the world do you write code so that it can be ported easily? C provides an excellent mechanism for conditional compilation, but this is only a small part of the solution.

How do you handle segmented versus non-segmented architectures? What about C and C++? There are slight differences between the two languages.

One solution that works really well is to abstract out the interdependencies between the environments into a set of macros so that the code base does not have to change.

Use macros as an aid to porting so that your code base does not change at all.

### 3.2.1 Segmented/Flat Architecture Macros

A number of #defines that provide a basis for further development are as follows.

```
Porting aids for Microsoft C8 segmented architecture
#define FAR _far
#define NEAR _near
#define FASTCALL _fastcall
#define PASCAL _pascal
#define EXPORT _export
#define BASEDIN(seg) _based(_segname(#seg))
```

#### **Porting aids for flat model programs**

```
#define FAR
#define NEAR
#define FASTCALL
#define PASCAL
#define EXPORT
#define BASEDIN(seg)
```

FAR and NEAR. These are used to abstract out the near- and far-segmented architecture. NEAR implies a 16-bit pointer and FAR implies a 32-bit pointer. When porting to a non-segmented architecture, these can be defined to be nothing.

FASTCALL and PASCAL. These are used to specify the calling convention of a function primarily for optimization purposes. When porting to a non-segmented architecture, these can be defined to be nothing.

EXPORT. This define is applicable to Windows DLL programming. Otherwise, it can be defined to be nothing.

BASEDIN. This define is primarily used by the CSCHAR macro to place character strings within a code segment primarily for optimization purposes. When porting to a non-segmented architecture, it can be defined to be nothing.

In most cases, these macros are used in other macros or in typedef's so that the code base is not cluttered up. For example, to declare a far pointer to a char so that it works equally well under a segmented and non-segmented architecture, you could do the following.

```
Using a far pointer to a char, not a good idea
char FAR*lpMem;
```

However, using char FAR\* will just clutter up all the source files with FAR. The solution is to use a typedef to declare what a far pointer to a char is once.

---

```
LPSTR typedef, a better idea  
typedef char FAR*LPSTR;
```

Now LPSTR should be used instead of char FAR\*. The concept of trying to hide how something works provides an abstraction that aids porting and allows for clean source code.

Try to hide how something works to provide an abstraction that aids

### 3.2.2 Using EXTERNC

If you are coding under C++, name mangling can sometimes be a problem. This happens under Windows DLL coding when a .def file is used. Functions that are exported must be specified in the .def file, but name mangling can make it almost impossible to type in the names manually. Luckily, C++ provides a solution in the form of a linkage specification. The #define's you can use are as follows.

```
EXTERNC macro  
/*--- EXTERNC depends upon C/C++ ---*/  
#ifdef __cplusplus  
#define EXTERNC extern "C"  
#else  
#define EXTERNC  
#endif
```

Under C++, EXTERNC gets defined to be extern "C". Under C, EXTERNC gets defined to be nothing. EXTERNC is used in function prototypes as follows.

```
Using EXTERNC  
EXTERNC type APIENTRY FunctionName( argument-types );
```

You need to use EXTERNC only in the prototype for a function, not in the source code where the function is actually written. This is how Microsoft C8 works.

### 3.3 Using WinAssert() Statements

The WinAssert() statement is the classic assertion statement with a few twists. Why use assertion statements? The key reason is to verify that decisions and assumptions made at design-time are working correctly at run-time. There is a difference between WinAssert() and [CompilerAssert\(\)](#) §2.1.4. Both check design-time assumptions, but CompilerAssert() provides the check at compile-time, whereas WinAssert() provides the check at run-time.


Assertion statements provide run-time checking of design-time assumptions.

**Microsoft C8 assert() macro, do not use**  
`#define assert(exp) \  
 ( (exp) ? (void) 0 : _assert(#exp, __FILE__, __LINE__) )`

An assert macro is provided by the Microsoft C8 library in assert.h. It takes any boolean expression. Nothing happens when the assert macro is true. If the assert macro is false, however, the \_assert() function is called with a string pointer to the text of the boolean expression, a string pointer to the text of the source file and an integer line number where the error occurred. Usage of the stringizing operator (#) is described in [§2.2.7](#). This information is then formatted and displayed by \_assert().

A problem with this macro is that it ends up placing too many strings in the default data segment. One easy solution is to remove the #exp argument, which is turning the boolean expression into text. After all, the file and line number are all that are needed to look up the boolean expression. Also, every time the assert() macro is used, a new string \_\_FILE\_\_ is created. Some compilers are able to optimize these multiple references into one

reference, but why not just fix the problem? My solution to the problem is to declare a short stub function at the top of each source file which references `__FILE__`. `WinAssert()` then calls this stub function with the current line number.

 There is an additional problem that is specific to the Windows programming environment. In Windows DLLs, it is possible to declare a function that, when called, does not switch to the DLLs data segment, but instead keeps the current caller's data segment. If you were to use an assertion statement in one of these functions, the string pointer to the filename would be incorrect. The solution is to declare the `__FILE__` string to be a code segment ([CSCHAR §2.1.8](#)) variable. This way, it does not matter what data segment is current.

An interesting twist that has been added to `WinAssert()` is that it supports writing code that is fault-tolerant. If a design-time assumption has failed, should you really be executing a section of code? I say no! The `WinAssert()` statement may be followed by a semicolon or by a block of code. The block of code will be executed only if the assertion succeeds.

Use `WinAssert()` on a block of code to produce code that is fault-tolerant.

**WinAssert(), non-fault-tolerant syntax**  
`WinAssert(expression);`

**WinAssert(), fault-tolerant syntax**  
`WinAssert(expression) {  
 (block of code)  
}`

The `WinAssert()` is implemented through a set of macros as follows.

**WinAssert() implementation**  
`#define USEWINASSERT CSCHAR szSRCFILE[]=__FILE__;` \

```

    BOOL static NEAR _DoWinAssert( int nLine ) {           \
        ReportWinAssert(szSRCFILE, nLine);                \
        WinAssert(nLine);                                  \
        return(FALSE);                                     \
    }
#define AssertError _DoWinAssert(__LINE__)
#define WinAssert(exp) if (!(exp)) {AssertError;} else

```

If WinAssert() is used in a source file, USEWINASSERT must appear at the top of the source file somewhere.

In addition to the WinAssert() macro, an AssertError() macro is provided for those times that you want to unconditionally force an error to be reported.

The reporting process of an assertion failure starts by calling a function that is local (private) to the source file. The function is \_DoWinAssert() and the argument is the line number where the failure occurred. The body of \_DoWinAssert() is straightforward except for the inclusion of WinAssert(nLine). Since the line number is never zero, this appears to have no purpose. This trick forces \_DoWinAssert() to be compiled into the module, even if there are no references to the function in the rest of the file. Otherwise, Microsoft C8 removes the unreferenced function.

Another subtle problem is that if \_DoWinAssert() is declared to be a LOCAL function (described in [Chapter 6](#)), the optimizing Microsoft C8 compiler will not build a stack frame for this function. For this reason, it has the static NEAR attributes instead of the LOCAL attribute, which allows the stack frame to be built.

In addition to these defines, WinAssert() requires that ReportWinAssert() be defined somewhere. I define it in a DLL so that the function needs to be coded only once.

```

ReportWinAssert() function prototype
EXTERNC void APIENTRY ReportWinAssert( LPSTR, int );

```



Once done, any other application has access to it. ReportWinAssert() allows you to display the assertion error in whatever way is appropriate at your organization. In my ReportWinAssert(), I log the filename, line number and stack trace to a log file and issue a system modal message box requesting that the user report the error. See [§A.7](#) for example implementations.

```
Using WinAssert()
#include <app.h>
USEWINASSERT
.
.
.
void LOCAL TestingWinAssert( int nValue )
{
    WinAssert(nValue>0) {
        ...
    }
} /* TestingWinAssert */
```

One of the key things you must remember is that the argument to WinAssert() must have absolutely no side effect on any variables. It must only reference variables.

```
WinAssert(), used incorrectly
WinAssert( (x/=2)>0 );

WinAssert(), used correctly
x /= 2;
WinAssert(x>0);
```

This is in case a policy of removing assertion statements from the code before releasing the product is enforced. While I do not recommend that you remove assertion statements, you still want to play it safe. You do not want to end up accidentally removing code that is needed to make your program run correctly.

## 3.4 Naming Conventions

One of the most important aspects of programming is the use of a consistent naming convention. Without one, your program ends up being just a jumble of various techniques and hence hard to understand. With a naming convention, your program is more readable and easier to understand and maintain.

I will describe the naming conventions that I have used to code a large application that have worked quite well for me.

### 3.4.1 Naming Macros

Macro names should always be in uppercase and may contain optional underscore characters. For macros that take arguments, I prefer not to use the underscore character anywhere (e.g., NUMSTATICELS()). For macros that define constant numeric values, underscore characters are OK (e.g., MAX\_BUFFER\_SIZE).

Macro names should be in uppercase.
-------------------------------------

Macro names in uppercase stand out and draw attention to where they are located. Some macros that are universally used throughout almost all code are allowed to be in mixed upper- and lowercase. An example of a macro like this is the [WinAssert\(\) macro §3.3](#).

There are many times that a set of macros contain a common subexpression. When this happens, I create another macro that contains the common sub-expression. The sole purpose of this type of macro is that it is to be used by other macros and not in the source code. A naming convention I use to help me remember that the macro is private to other macros is to name it with a leading underscore character.

Macros beginning with an underscore are to be used only in other macros, not explicitly in source code.
---

### 3.4.2 Naming Data Types

I can remember the difficulty I had coming up with good data type names when I first started to code. I was using mixed upper- and lowercase for data type names and variable names. However, it became harder and harder to read the program. I always ended up wanting the variable name to be spelled the same as the data type name but could not do this, so I ended up calling it something different which made the program hard to understand.

The convention that I finally settled upon is that all data types should be in uppercase. The variable names can then be spelled the same, but in mixed upper- and lowercase. This convention may at first seem awkward, but in practice I have found that it works well.

New data types should be in uppercase.

Data type names should also avoid using the underscore character. This is because macro names may use the underscore character and it is best to avoid any possible confusion or ambiguity over whether or not an uppercase name is a data type name or macro name.

### 3.4.3 Declaring Data Types

New data types must be declared with a typedef statement. While it is possible to use a macro to create what looks like a new data type, it is not a true data type and is subject to subtle coding problems.

New data types must be declared with a typedef statement, not a macro definition.

Consider the data type PSTR, shown below, which is a character pointer.

**Using typedef to create a new data type**  
`typedef char*PSTR;`

**Using macros to create a new data type, a bad practice**

```
#define PSTR char*
```

#### Using the new PSTR data type

```
PSTR pA, pB;
```

In the above example, what is the type of pA and what is the type of pB? In the case of using typedef to create the new data type, the type of pA and pB is a character pointer, which is as expected. However, in the case of using the macro to create the new data type, the type of pA is a character pointer and the type of pB is a character. This is because PSTR pA, pB really represents char \*pA, pB which is not the same as char \*pA, \*pB.

This example shows the danger in using macros to declare new data types in the system. Therefore, you should avoid using macros to declare new data types.

### 3.4.4 Naming Variables

All variables should be named using the Hungarian variable naming convention with mixed upper- and lowercase text and no underscore characters.

Variables should be named using Hungarian notation.

The Hungarian naming convention states that you should prefix all variable names with a short lowercase abbreviation for the data type of the variable name. (See Table 3-1).

Prefix	Data Type
a	array of given type
b	BOOL: true/false value
by	BYTE
c	char
dw	DWORD

h	handle or abstract pointer
l	long
lp	long pointer
n	int
p	pointer
w	WORD


Table 3-1: Hungarian Notation Prefixes

For example, nSize is an integer, bOk is a BOOL and hIcon is an abstract handle. Prefixes may be combined to produce a more descriptive prefix. An example would be lpnCount, which is a long pointer to an integer and lpanCounts, which is a long pointer to an array of integers.

The advantage of Hungarian notation is that you are much more likely to catch a simple programming problem early in the coding cycle, even before you compile. An example would be nNewIndex = lIndex+10. Just by glancing at this you can see that the left-hand side is an integer and the right-hand side is a long integer. This may or may not be what you intended, but the fact that this can be deduced without seeing the original data declarations is a powerful concept.

Hungarian notation allows you to know a variable's data type without seeing the data declaration.

The Hungarian notation handles all built-in data types, but what about derived types? A technique that I have found useful is to select an (uppercase) data type name that has a natural mixed upper- and lowercase name.

 An example from the Windows system is the HICON data type and the hIcon variable name. As another example, let's consider a queue entry data type called LPQUEUEENTRY. A variable name for this could be lpQueueEntry.

This convention works great for short data type names like HICON, but not

so well for long data type names like LPQUEUEENTRY. The resulting variable name lpQueueEntry is just too long to be convenient. In this case, an abbreviation like lpQE should be used. However, make sure that lpQE is not also an abbreviation for another data type in your system.

Whatever technique you use to derive variable names from data type names is fine provided that there is only one derivation technique used in your entire program. A bad practice would be to use lpQE in one section of code and lpQEntry in another section of code.

A data type must have a single and unique variable derivation.
--

### **3.4.5 Naming Functions**

Functions should be named using the module/verb/noun convention in mixed upper- and lowercase text with no underscore characters.

Functions should be named using the module/verb/noun convention.
--

Suppose you have just started a project from scratch. There is only one source file and the number of functions in it is limited. You are naming functions whatever you feel like and coding is progressing rapidly. Two months go by and you are working on a new function that needs to call a specialized memory copy routine you wrote last month. You start to type in the function name, but then you hesitate. Did you call the function CopyMem() or MemCopy()? You do not remember, so you look it up real quick.

This actually happened to me and the solution was simple. Follow the Microsoft Windows example of naming functions using the verb/noun or action/object technique. So, the function should have been called CopyMem().

This solved my immediate problem, but not the long-term problem. It wasn't long before I had thousands of function names, some with similar sounding verb/noun names. My solution was to prefix the verb/noun with the module name.

Suppose you have a module that interfaces with the disk operation system of your environment. An appropriate module name would be `Dos` and several possible function names are `DosOpenFile()`, `DosRead()`, `DosWrite()` and `DosCloseFile()`.

Module names should contain two to five characters, but an optimum length for the module name is three to four characters. You can almost always come up with a meaningful abbreviation for a module that fits in three to four characters.

### **3.5 Chapter Summary**

- A rock-solid layer needs to be built upon all system level interfaces because system level interfaces contain bugs. You cannot assume that they are bug-free. I have been burned too many times to trust system level calls blindly.
- When you do make assumptions in calling system code, it is best to `WinAssert()` these situations in a code wrapper. When a function does fail and you assumed that it never would, you want to know about it so that you can fix the problem and reevaluate your assumption.
- Macros can be used as an abstraction layer to allow your code to be ported without having to change your source files. Instead, just change the macros contained in your include files and recompile. The macro name in the source file is specifying what to do. The macro body in the include file is specifying how to do it.
- It is a good idea to use `WinAssert()`'s generously in your code. The `WinAssert()` provides run-time checking of design-time assumptions and signals a design flaw when it occurs. Use the fault-tolerant form of `WinAssert()` whenever possible.
- When programming in a large project, it is crucial that a consistent naming convention be used throughout the entire project. This helps prevent misunderstandings that produce buggy code.
- Macro names should be in uppercase. A macro beginning with an underscore character is intended to be used only in other macros, not explicitly in source code.

- New data types should be in uppercase and declared with a typedef statement, not a macro definition.
- Variables should be named using the Hungarian notation. This notation allows you to know the type of a variable without seeing the data declaration.
- Functions should be named using the module/verb/noun convention.

---

Copyright © 1993-1995, 2002-2003 Jerry Jongerius  
This book was previously published by Person Education, Inc.,  
formerly known as Prentice Hall. ISBN: 0-13-183898-9



# Chapter 8: Style Guide [Writing Bug-Free C Code](#)

[8.1 Commenting Code](#)

[8.2 Indentation Rules/Examples](#)

[8.3 Defining Auto Variables](#)

[8.4 Avoid Using the Comma Operator](#)

Writing a style guide is hard because every programmer has his or her own way of doing things. Despite this problem, I feel that it is important that a style guide exist in written form at any organization. The rest of this chapter describes the style that I use to write code.

## 8.1 Commenting Code

Every module and every function has a comment header. The layout of a module is covered in [Chapter 6](#).

Within a function there are two basic forms of comments that I use. They are the Endline comment and the Inline comment.

### 8.1.1 Endline Comments

Endline comments are used almost exclusively to document variable declarations. These declarations can be auto variable definitions or structure member declarations.

An endline comment uses the C++ style single line comment `//` (two slashes) and begins in column 41.

**EndLine comment examples, using C++ //**

```
typedef struct {  
    int x;                // the x coordinate of the point  
    int y;                // the y coordinate of the point  
} POINT;  
  
int main(void)  
{  
    int nCpuBusy;         // how busy is the cpu  
    (code body)
```

```
    return 0;
}
```

If you do not have access to a C++ compiler, use the standard C `/* */` comment form as follows.

```
EndLine comment examples, using /* */
typedef struct {
    int x;                /* the x coordinate of the point */
    /*
    int y;                /* the y coordinate of the point */
    */
    } POINT;

int main(void)
{
    int nCpuBusy;         /* how busy is the cpu */
    (code body)
    return 0;
}
```

### 8.1.2 Inline Comments

The inline comment is used to comment a small section of code within a function.

```
Inline comment example
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
    static char buffer[9000]; /* buffer for input line */
    long lLineCount=0;        /* number of lines read */
    /*--- Count the number of lines in the input ---*/
    while (safegets(buffer,sizeof(buffer))) {
        ++lLineCount;
    }
    /*--- Display the total line count ---*/
    printf( "Number of Lines = %ld\n", lLineCount );
    return 0;
}
```

```
} /* main */
```

The inline comment starts with `/*---` and ends with `---*/` and always has one blank line preceding it.

A technique that I use when writing the inline comments of a function is to pretend that all the code in the function has been removed. Would reading the remaining inline comments be helpful and meaningful? Try to write the inline comments so that they would make sense in this context.

## 8.2 Indentation Rules/Examples

All of my indentation levels are based on four spaces. So the first indentation level starts in column 5, the next indentation level starts in column 9, and so on.

### 8.2.1 Functions

A function must always consist of a comment header followed by a blank line and the function body.

#### Function template

```
/*pf-----  
-----  
 *  
 *  DESCRIPTION: (Short Description)  programmers-initials  
 *  
 *    (A long description of the function)  
 *  
 *  ARGUMENTS:  
 *  
 *    Arg1 - Arg1 description  
 *    ...  
 *    ArgN - ArgN description  
 *  
 *  RETURNS:  
 *  
 *    (A description of the return value)  
 *
```

```

*   NOTES:
*
*   (optional notes section)
*
*-----
--*/

type APIENTRY FunctionName( arguments )
{
    type variable1;           /* Comment */
    type variable2;           /* Comment */
    /*--- Comment ---*/
    (block of code)
    /*--- Comment ---*/
    (block of code)
    ...
} /* FunctionName */

```

The begin and end brace of a function are on separate lines in column 1. In addition, the ending brace is followed by a space and a comment that is the function name.

Local variable declarations and the main code in the function are at the first indentation level. The local variables of the function, if any, are declared first and always have endline comments.

The code within the function uses inline comments to document the code and follows the blank line before an inline comment rule. There is always one blank line after the last line of code and before the ending brace of the function.

### 8.2.2 The IF Statement

The if statement begins with one line containing the if statement, the expression and the begin brace. There is one space after the if statement and before the begin brace. The body of the if statement and the ending brace are at the next indentation level.

**if statement template**

```
/*--- Comment ---*/  
if (expression) {  
    statement;  
    ...  
}
```

The if statement always has a begin brace and an end brace, even when followed by only a single statement.

### 8.2.3 The IF/ELSE Statement

The if statement begins with one line containing the if statement, the expression and the begin brace. There is one space after the if statement and before the begin brace. The body of the if statement and the ending brace are at the next indentation level.

The else statement begins at the same indentation level as the if statement and is followed by a space and a begin brace. The body of the else statement and the ending brace are at the next indentation level.

```
if/else statement template  
/*--- Comment ---*/  
if (expression) {  
    statement;  
    ...  
}  
/*--- Else comment ---*/  
else {  
    statement;  
    ...  
}
```

The if statement and the else statement always have a begin brace and an end brace.

### 8.2.4 The WHILE Statement

The while statement begins with one line containing the while statement,

the expression and the begin brace. There is one space after the while statement and before the begin brace. The body of the while statement and the ending brace are at the next indentation level.

```
while statement template  
/*--- Comment ---*/  
while (expression) {  
    statement;  
    ...  
}
```

The while statement always has a begin brace and an end brace.

### 8.2.5 The DO/WHILE Statement

The do statement begins with one line containing the do statement, a space and the begin brace. The rest of the do/while statement is at the next indentation level. The do/while ends with an end brace, the while statement and the expression on one line. There is one space after the ending brace and after the while statement.

```
do statement template  
/*--- Comment ---*/  
do {  
    statement;  
    ...  
} while (expression);
```

### 8.2.6 The FOR Statement

The for statement begins with one line containing the for statement, the three expressions and the begin brace. There is one space after the for statement and before the begin brace. The three expressions are contained in parentheses and are separated by a semicolon and space.

```
for statement template
```

```
/*--- Comment ---*/  
for (loop=0; loop<nMax; ++loop) {  
    statement;  
    ...  
}
```

The for statement always has a begin brace and an end brace.

### 8.2.7 The SWITCH Statement

The switch statement begins with one line containing the switch statement, the expression and the begin brace. The body of the switch statement and the ending brace are at the next indentation level.

The body of the switch statement contains case statements and a default statement. Each case and default are on a new line with the body of each case and optional default at a new indentation level.

```
switch statement template  
/*--- Comment ---*/  
switch (expression) {  
    case constant:  
        statement;  
        ...  
        break;  
    case constant: {  
        statement;  
        ...  
        break;  
    }  
    default:  
        statement;  
        ...  
        break;  
}
```

If a case uses local variables, begin and end braces are used. The begin brace is on the same line as the case and begins after the colon followed by a space. The end brace is on a new line after the break at the same

indentation level as the body of the case.

## 8.3 Defining Auto Variables

Auto variables should be defined one per line. Do not use the comma to define multiple variables per line. Consider the following.

```
Bad use of comma in auto variable definition
void Testing( void )
{
    char *pBeg, pEnd;           /* comment */
    ...

} /* Testing */

Correct way to define auto variables
void Testing( void )
{
    char *pBeg;                 /* comment */
    char *pEnd;                 /* comment */
    ...

} /* Testing */
```

The problem with this code is that the data type of pBeg is char\* and the data type of pEnd is char, which is more than likely not what was intended. The preferred way to define auto variables is one per line.

## 8.4 Avoid Using the Comma Operator

The comma operator is typically used to treat two expressions as one. This can be useful in situations where only one expression is allowed. The value of the two expressions separated by the comma operator is the value of the second expression.

```
Comma operator syntax
expression, expression
```



However, using the comma operator can lead to code that is hard to read and maintain. It is best to avoid using it.

One possible exception is in macros where it is important that the macro be viewed by the programmer as a single expression. If you cannot accomplish everything in one expression, do you give up? In cases like this, it is important to avoid having to change the syntax of how a macro is used and usage of the comma operator is allowed.

# Chapter 9: Conclusion [Writing Bug-Free C Code](#)

[9.1 The Class Methodology](#)

[9.2 Run-Time Type Checking](#)

[9.3 Isolating Change through Macros](#)

[9.4 The Learning Process Never Stops](#)

The [class methodology \(Chapter 4\)](#) is the core methodology described in this book. It makes writing C code easier because it solves the information overload problem that occurs when too many data structures are declared in include files.

## 9.1 The Class Methodology

The class methodology moves data structure declarations out of include files and places them instead in class implementation modules. Data structures are never accessed directly by code that uses a class. Instead, data structures are turned into private objects that are controlled by calling method functions -- functions that create, manipulate and destroy an object.

All the code that manipulates an object is now isolated into one source file (or module). This leads to a program that consists solely of a number of well-isolated modules. Such a program is easy to enhance and maintain, since changing the implementation of an object involves only code changes in one source file.

The method function names are specifying what to do, not how to do it. For example, `DosWriteFile(hDosFh, lpMem, wSize)` is specifying that we want to write some information to a file. The how is left up to `DosWriteFile()`.

Applying the class methodology to a program is a deceptively simple thing to do, but it is a powerful concept when applied to an entire program.

## 9.2 Run-Time Type Checking

If all method functions of a class employ run-time type checking on pointers (handles) passed into the module, a lot of common programming

errors will be detected automatically and reported.

When combined with full symbolic stack traces at the point of failure, almost all problems can be deduced from the symbolic stack trace and fixed. A problem does not have to be reproducible in order to track down the problem.

### **9.3 Isolating Change through Macros**

Over the years, the class methodology has undergone a lot of changes in how it is implemented. However, through all these changes, the code base has changed little.

The macros that were used in the source code specify what to do and not how to do it. For example, the NEWOBJ(hObj) and VERIFY(hObj) macro syntax has stayed the same, but the implementation of these macros has changed drastically.

The key is to pick the correct what (or interface). If done correctly, the what can stay the same and the how can change drastically.

### **9.4 The Learning Process Never Stops**

Writing bug-free C code takes a lot of effort. It is not something that just happens. You could have all the latest whiz-bang tools and languages, but they do not help you to write bug-free code unless you have a thorough knowledge of the tools and languages themselves.

Consider a carpenter's tools. If you were given all of the carpenter's tools, could you build a house? Of course not. Why? Mainly because you do not have the knowledge of how to use the tools. The same goes for programming and writing bug-free code. How can you be expected to write quality code unless you know your tools inside out? No matter what skill level you are at, there will always be something new to learn because the learning process never stops. I am amazed that even after years of programming in C I am still learning new nuances about the language.

I have no doubt that the techniques described in this book will continue to be refined. I would be disappointed if they were not. They are just a snapshot of the techniques that I use today, techniques that have been refined over many years of programming in C.

I hope you have learned from my techniques something new about writing bug-free C code.

---

Copyright © 1993-1995, 2002-2003 Jerry Jongerius  
This book was previously published by Person Education, Inc.,  
formerly known as Prentice Hall. ISBN: 0-13-183898-9

## References [Writing Bug-Free C Code](#)

- (1) Hummel, Robert L, **Programmer's Technical Reference: The Processor and Coprocessor**. Ziff-Davis Press, Emeryville, CA. 1992. Has an interesting chapter on bugs in the 80x86 chip family. A must for anyone using 80x86 assembly in their programs.
- (2) Kernighan, Brian W., and Dennis M. Ritchie. **The C Programming Language**. Prentice Hall, Englewood Cliffs, NJ. Second Ed. 1988. The bible for C programmers. It is written by the original designers of C and it should be a part of every C programmer's library.
- (3) Petzold, Charles. **Programming Windows**. Microsoft Press, Redmond, WA. Second Ed. 1990. A good introduction to Windows programming. If you program in Windows, you need this book.
- (4) Pietrek, Matt. Windows Internals. **The Implementation of the Windows Operating Environment**. Addison-Wesley, Reading, MA. 1993. A great book on the internals of Windows, giving insight into how Windows works.
- (5) Schulman, Andrew, David Maxey, and Matt Pietrek. **Undocumented Windows**. A Programmer's Guide to Reserved Microsoft Windows API Functions. Addison-Wesley, Reading, MA. 1992. A great book on the undocumented internals of Windows, but it is not for the beginner or fainthearted.
- (6) Shaw, Richard Hale. **"Based Pointers: Combining Far Pointer Addressability and the Small Size of Near Pointers,"** Microsoft Systems Journal, September 1990, p. 51. A useful article on the use of based pointers.
- (7) Stroustrup, Bjarne, and Margaret A. Ellis. **The Annotated C++ Reference Manual**. Addison-Wesley, Reading, MA. 1990. Known as the ARM, this book is the ANSI base document for C++. A book for the serious C/C++ programmer, it is filled with insightful commentary sections.

(8) Stroustrup, Bjarne. **The Design and Evolution of C++.**

---

Copyright © 1993-1995, 2002-2003 Jerry Jongerius  
This book was previously published by Person Education, Inc.,  
formerly known as Prentice Hall. ISBN: 0-13-183898-9