Metagraphics C/C++ Programming Guidelines

Version 4.3 – January 3, 2002

Metagraphics Corporation
PO Box 225
Woodinville, WA 98072

Phone:	425-844-1110
Fax:	425-844-1112
Web:	http://www.metagraphics.com

Document:	http://www.metagraphics.com/pubs/MetagraphicsCodingGuide.pdf (.doc)
Author:	Jack Davis <jack@metagraphics.com>
Email:	support@metagraphics.com

CONTENTS

1 Naming Conventions

The proper naming of items is a foundation for building well documented and maintainable code. Thoughtfully chosen names aid in documenting the code and lessen the need for explicit comments (that's not to say that good commenting isn't still very important!).

1.1 Use Common Word Names

For constant and variable identifiers try to use names that are built from common English words, and that clearly describe the item being defined. Avoid abbreviations where possible, they usually just hurt readability.

Commonly used abbreviations are an exception:

`col`	column index
`cur`	current
`i j k`	generic counters
`x y z`	Cartesian coordinates
`min`	minimum
`max`	maximum
`src`	source
`dst`	destination
`num`	number of
`p ptr`	pointer
`s str`	string

1.2 If You Must Use Hungarian…

While Microsoft seems to be committed in promoting Hungarian notation on the rest of world, there seems very little real benefit gained from its usage. One of the main negatives with Hungarian notation is that the name mangling severely hurts readability. For example, something simple like:

```
char *str;
```

is declared in Hungarian notation as:

```
LPCSTR lpszstr;
```

`lpszstr` translates to: "long pointer to string terminated by zero called `str`". (Is this really easier to understand?)

Even the use of the "p" prefix to designate pointers is usually of marginal benefit, as in the following example:

```
RECT *pRect = malloc( sizeof(RECT) );
pRect->xmin = 0;
```

In reading the above statements, it's obvious from the usage that `pRect` is a pointer – stating it twice seems redundant. Keeping things simple is usually best:

```
RECT *rect = malloc( sizeof(RECT)  );
rect->xmin = 0;
```

If you need both an object and a pointer to the object in the same scope, then a "p" pointer prefix might be useful:

```
char  str[256];
char *pstr = str;
```

In many cases, however, using a different descriptive name can be better:

```
char  str[256];
char *firstNonWhite = str;     /* better than "pstr" */

while ( isspace( *firstNonWhite) )
    firstNonWhite++;
```

Hungarian Prefixes

Hungarian notation in constant and variable names is not required (nor generally recommended). If however a project does decide to use Hungarian notation, the following standards should be used for consistency.

Modifiers		Basic Types	
		c	char
		i	int
l	long	b	BOOL
p	pointer	f	float
d	double	s	char[]
u	unsigned	sz	char[] w/NULL
s	short	h	handle
		fn	function
		t	structure

Examples

```
char           cMI;                /* a single character              */
int            iPitch;             /* a signed integer                */
unsigned int   uRowBytes;          /* an unsigned integer             */
double float  *pdfDistance;        /* pointer to a double             */
unsigned char  ucCount;            /* an unsigned 8-bit counter       */
char           szInputString[64];  /* a NULL terminated string        */
char          *pszInputString;     /* pointer to NULL terminated string */
```

1.3 Types & Constants

The names of typedefs and constants are normally all uppercase. Underscores may be used to improve readability. To reduce potential naming conflicts, a two-character prefix should be used for library types and constants. Here are some examples:

```
typedef unsigned char  UCHAR
typedef int  BOOL
#define MAX_BRAINS  100
#define MWCONFIG_BITMAPSUPPORT_8BIT
```

1.4 Constants & Macros

Constants – Use `enum` or `const` Instead of `#define`

Avoid using `#define` for symbolic constants. Instead, use either `enum` or `const`. Unlike `#define` symbols, `enum` and `const` follow C scope rules and have types associated with them.

```
#define cGREEN 1                    /* poor   */
const int cGreen = 1;               /* good   */
enum { cGreen = 1 };                /* better */
enum Color { red, blue, green };    /* best   */
```

`#define` can be especially dangerous if someone redefines the name – the compiler will blindly go off and change all related references in your program. With `enum` or `const` you get an error message. An `enum` offers several special advantages:

1. With an enumeration identifier (such as "Color", above), each identifier is treated as a separate type for the purpose of type checking.
2. An enum, like a #define, does not allocate any memory and can be placed in a header file. A C const definition such as "const int num=123;" on the other hand actually allocates space for the int and initializes it to the specified value. As such, a const definition shouldn't be used in a header file (unless you make it a global variable using extern – *not recommended!*).
3. An enum can be used as the argument of a case statement (a const cannot).
4. An enum can be used as the array size in an array declaration (a const cannot).
5. Both enum and const allow you to limit identifiers to the scope of a specific function or class (a #define cannot).
6. enum and const reduce simple mistakes. Here's a classic problem with #define:

```
#define VALUE1 512
#define VALUE2 VALUE1+1024
```

later in the code

```
n = 10 * VALUE2;
```

which is evaluated as:

```
n = (10 * VALUE1) + 1024;  /* OPPS! - this isn't the answer we expect! */
```

Adding some parenthesis in the #define will correct this problem, but enum and const eliminate these kind of mistakes.

Macros – Use inline Instead of #define for Function Macros

Using a #define for a function macro can be dangerous if the parameters are used more than once. For example, given the following macro definition, the later code statement executes differently than expected:

```
#define Square(x) ((x)*(x))
```

later in the code:

```
z = Square( x++ );
```

In the above example, z is computed properly but x is incremented twice instead of once as would be expected. For this type of situation an inline function works correctly:

```
inline long Square( short x )
    { return ( (long)x * (long)x ); }
```

#define Constants and Macros

Avoid using #define's where possible, instead using the preferred enum or const forms for general constants, or inline for macros functions. When used, #define constants and macros should be in all UPPERCASE. Underbars "_" may be inserted to separate words for clarity.

```
#define OUT_OF_MEMORY 0x0010
#define DEBUGSTRING (s) printf (s)
```

You need to be careful with #define's that are defined as part of a shared library. To avoid conflicts with application names and other system names, #define's used within a library should be prefixed by a two uppercase character library identifier.

```
#define MWERROR_FILE_IO             16     /* Metagraphics MetaWINDOW library */
#define MLERROR_INVALID_FILE_FORMAT 0x0008 /* Metagraphics Media!Lab library  */
#define TSERROR_INVALID_FONT_FILE   0x000A /* Metagraphics TypeServer library */
```

1.5 Variables

Except for local loop index variables (such as `i`, `j`, etc.), variable names should be at least three characters long, not counting any prefixes. Variable names should not contain underscores except where specified in a prefix.

Global Variables

Global variables should begin with the prefix "g_" with the first letter of all words in the variable name capitalized:

```
int  g_PixelsWide;
int  g_PixelsHigh;
```

In general, global variables should be avoided whenever possible. Global variables impact the global name space and can create coupling relationships and nasty maintenance problems. When used, global variables should be initialized with their declaration, or be self-initializing as part of an associated internal function structure.

Member Variables

Member variables of a C++ class should begin with the prefix "m_" with the first letter of all words in the variable name capitalized:

```
int  m_PixelsWide;
int  m_PixelsHigh;
```

Static Variables

For static variables global to a single source file (not static local variables, see below),should begin with the prefix "s_" with the first letter of all words in the variable name capitalized:

```
static int  s_PixelsWide;
static int  s_PixelsHigh;
```

In a C and C++ source file, static variables should be defined at the beginning of the source file after the header file `#include's` and before the static function prototypes.

Local Variables

For local variables (including static local variables), the first word should be all lowercase and subsequent words should capitalize the first letter:

```
int        pixelsWide;
int        pixelsHigh;
static int rowBytes;
```

Function Arguments

Similar to local variables, local function arguments should begin with the first word entirely lowercase and subsequent words with the first letter capitalized:

```
BOOL ReadAudioHeader( UINT headerSize );
LONG ReadAudio( void *buffer, LONG bufferSize );
LONG ReadFrame( int dstX, int dstY, const RECT *dstRect );
```

1.6 Function and Class Names

Function names should be mixed case with first letter of each word of the name capitalized. Function names should not contain underscores.

Avoid Microsoft and ANSI C Names

Microsoft Windows currently defines some 1300+ functions, over 80 classes and an immense number of `#define's`. ANSI C also reserves symbols starting with an underscore "_" and type names ending with `_t`. These name spaces can also be expected change and grow. If you pick a name that ANSI C or Microsoft uses, or chooses to use later, somebody is going to have to change their code (and it's likely not Microsoft).

Avoiding ANSI names is simple, other than the variable-name prefixes `g_` (global), `s_` (static) and `m_` (member) outlined earlier, don't use underbars in any functions or class names. You need to be more careful to avoid Microsoft name conflicts. To be safe a project may optionally choose a two-character uppercase identifier to prefix its function and class names. Names in shared C function libraries and shared C++ class libraries should use a two-character lowercase prefix to avoid conflicts with application and system names (see below).

C Application Functions

Function names for C application and system functions should be one or more words with first letter of each word capitalized. Function names should not have underscores.

```
void PaintWindow();
void ViewProperties( HWND hWnd );
void SetDialogPageItems( HWND hDlg, int dlgPage );
void ErrorMessage( HWND hWnd, char *messageString );
```

C Library Functions

To avoid system or application conflicts, names used for functions in a shared C library should begin with a two lowercase character library identifier followed by a mixed case name with the first letter of each word in the name capitalized. Library function names should not contain underscores.

For example, Metagraphics C library functions use the following prefixes:

`mg`	Metagraphics (generic)
`mw`	MetaWINDOW
`ml`	Media!Lab
`mx`	Media!FX
`mk`	Media!Key
`ts`	TypeServer

Sample Metagraphics library functions:

```
void    mwMoveTo( int x, int y );
MRESULT mkValidateName( char *firstName, char *lastName );
void    tsGetMetrics( TSMETRICS *metrics );
```

C++ Application Classes

All application and system classes should begin with the prefix "C", followed by one or more words with the first letter of each word capitalized. Class names should not contain underscores.

Sample application and system class names:

```
class CMfcPlayApp
class CString
class CListNode
class CLinkedList
```

C++ Library Classes

To avoid system and application class name conflicts, names for classes in a shared C++ class library should begin with a three-character prefix. The class library prefix should begin with a lowercase two-character library identifier followed by a capital letter "C". For example, Metagraphics C++ library classes use a prefix of "mgC". (Fewer prefixes are needed for C++ since a small group of classes usually encapsulate a larger list of functions.) Library class names should not contain underscores.

Sample Metagraphics C++ class names:

```
class mgCAudioWAV : virtual public mgCFile, virtual public mgCAudio
class mgCBitmap   : virtual public mgCObject
class mgCImageAVI : public mgCAudioWAV, public mgCImage
```

C++ Class Methods

Since C++ class methods are qualified within the scope of a specific class, C++ class methods do not need any special prefixes even when part of a shared class library (for class libraries, the class name itself should include a library prefix, as noted above). All C++ class methods should be mixed case with first letter of each word in the name capitalized. Class method names should not contain underscores.

```
BOOL ReadAudioHeader( UINT headerSize );
LONG ReadAudio( void *buffer, LONG length );
LONG ReadFrame( int dstX, int dstY, const RECT *dstRect );
```

1.7 Summary

The naming conventions outlined above allow someone reading the code to quickly discern where the definition of an object is located. Using these conventions simplifies identifying the object type and the location where it's defined.

`g_PixelsWide`	a global variable, global across multiple source files.
`m_AudoInfo`	a member variable within a class.
`s_PixelsHigh`	a static variable, global to this source file only.
`rowBytes`	a local variable or local function argument.
`GREEN`	an application `#define` constant or macro.
`MWERR_CODE`	a shared library `#define` constant or macro.
`PaintWindow()`	an application C function.
`mwMoveTo()`	a shared C library function
`CString`	an application or system class
`mgCBitmap`	a shared C++ library class
`.ReadFrame()`	a C++ class method.

2 Basic Data Types

ANSI C notes that the size of standard types such as `int, short, long, float, double`, etc. are machine and/or compiler dependent. One of the only general guaranteed relationship is that:

```
sizeof(short) <= sizeof(int) <= sizeof(long)
```

For our purposes, we can assume an `int` or a `short` is a minimum of 16-bits, and a `long` is a minimum of 32-bits. Any of these may be larger, however, depending on the target platform and compiler. For consistency and portability, you should use the definable data types noted below:

Data Type	Win32 Data Type	Description
BYTE	`unsigned char`	8-bit unsigned integer (0 to +255)
CHAR	`signed char`	8-bit signed integer (-128 to +127), or ANSI character
⇔ **INT**	`int`	16- or 32-bit (or future 64-bit) native signed integer
INT8	`signed char`	8-bit signed integer (-128 to +127)

`INT16`	`short`	16-bit signed integer (-32768 to +32767)
`INT32`	`long`	32-bit signed integer (-2,147,483,648 to +2,147,483,647)
`INT64`	`(future)`	64-bit signed integer (a very big signed number)
⬄ **`LONG`**	`long`	32-bit (or future 64-bit) long integer
`SHORT`	`short`	16-bit signed short integer (-32768 to +32767)
`TCHAR`	`char or INT16`	8-bit or 16-bit character, dependent if "_UNICODE" is defined
⬄ **`UINT`**	`unsigned int`	16- or 32-bit (or future 64-bit) native unsigned integer
`UINT8`	`unsigned char`	8-bit unsigned integer (0 to +255)
`UINT16`	`unsigned short`	16-bit unsigned integer (0 to +65535)
`UINT32`	`unsigned long`	32-bit unsigned integer (0 to +4,294,967,295)
`UINT64`	`(future)`	64-bit unsigned integer (a very big unsigned number)
⬄ **`ULONG`**	`unsigned long`	32-bit (or future 64-bit) unsigned long integer
`USHORT`	`unsigned short`	16-bit unsigned integer (0 to +65535)

Preferred types indicated in **`BOLD`**.
⬄ Indicates machine-dependent variable size data types.

Variable Size Data Types

An `INT` will typically be a native size for the target machine, and is usually the optimum size for fastest integer operations (`INT`'s are 64-bits on the new 64-bit processors!). Other sizes, either large or smaller, may incur a performance penalty (a `short` on a 32-bit Pentium processor, for example, requires an instruction prefix byte which slows execution).

It's best to use `INT` or `UINT` for local variables that can be contained within a minimum 16-bit value. For local integer variables that require more than 16-bits, use `LONG` or `ULONG`. `LONG` and `ULONG` variables will be a minimum of 32-bits, but similarly may increase in size, again dependent on the target platform (for 64-bit processors `LONG` and `ULONG` may either be 32- or 64-bits).

`INT`, `UINT`, `LONG` and `ULONG` are best used for local run-time variables for optimal performance. *Since these types may change in size dependent on the target machine, they should **not** be used in any data structure or data type that is written to a file or communicated to another machine!*

Fixed Size Data Types

For data structures and objects that are to be written or communicated externally, you should use the fixed size data types: `CHAR`, `BYTE`, `INT16`, `UINT16`, `INT32` or `UINT32`. These types are guaranteed to be their specified size.

3 Formatting and Documentation

Code without clear comments and through documentation is next to worthless (or in many cases, less than worthless). Thorough documentation saves substantially in not only in later maintenance and support, but also in up front development. By putting into words what the code logic is doing, you perform a double check on the logic in your design.

3.1 Write Descriptive Comments In Blocks

Comments are generally best if placed in multiline blocks alternating with blocks of code. This describes at a high level what the next section of code is doing. If the code section is complex, you can use a form of footnote comments to identify specific code areas:

```
/* Here is a block comment describing the block of code that follows.
 * After a general summary, we describe the specifics:
 *
 *   1. This comment describes what's happening at the line labeled <1>.
 *
 *   2. This comment describes what's happening at the line labeled <2>.
 */

here_is_the_code();
while ( some_condition )
{
   this_code_is_rather_obscure();      /* <1> */
}
more_stuff_here();
while ( some_condition )
{
   this_code_is_also_obscure();        /* <2> */
}
```

3.2 Document Defensively

Document In More Detail Than You Think You Need

When you're in the middle of coding a particular function or algorithm, you usually have a good understanding of its design and it's clear (to you) how it works. Looking at the same code six months or a year later, the design and operation will not be as obvious. Document in more detail than you think you need. Thoroughly describe the overall design and operation of your code. Documenting even the simpler (at the time) details will help you or someone else later to get back up to speed faster if you need to review or make changes to the code.

The level of detail to which to document is, of course, somewhat subjective – use your best judgment. On the other extreme, you needn't go overboard in documenting the obvious (such as "`x++; /* increment x */`" – comments like this just add clutter to the code). Be defensive though – if there's a doubt, write it out!

Use An External .doc File If Necessary

For documentation describing the development of algorithms, especially if more complex algebraic expressions or notations are involved, use a separate Word .doc file. Word files provide enhanced features such as the MS Equation Editor to help you more clearly write your documentation. Word files also allow you to imbed graphics and illustrations to make things even clearer (remember "one picture is worth a thousand …").

If you use an external .doc file, try to use the same main filename as the related .c or .cpp source code file. Also insert a comment in the source code file referencing the external documentation file. Typically .doc files are kept together in a separate `\doc` directory adjacent to the `\src` source code directory for the project. Make sure that all external documentation files are also checked into source control.

3.3 Align Comment Blocks Vertically

When writing C style comment blocks, align the `/*` and `*/` vertically in multiline comments:

```
/* First line,
 * second line,
 * third line,
 */
```

Here's an example of a comment block that can be improved:

```
/*********************************************************************

void a_function( void )

    Here is a multiline comment, doing all the great
    things a multiline comment should do.

    Unfortunately the lack of a vertical line of stars to the left
    makes it difficult to visually separate the comment from the code

*********************************************************************

void a_function( void )
{
    /* here is the actual function */

    code_goes_here();
}

/*********************************************************************/
```

In addition to making it visually difficult to see what is the comment and what is the code, it's easy to lose and hard to spot a missing closing `*/`. Here's the preferred format:

```
/*********************************************************************
 *
 * void a_function( void )
 *
 *   Here is a multiline comment, doing all the great
 *   things a multiline comment should do.
 *
 *   Here the vertical line of stars to the left makes it
 *   easy to visually separate the comment from the code.
 *
 *********************************************************************
 */

void a_function( void )
{
    /* here is the actual function */

    code_goes_here();
}
```

For C++ simply use `//` to begin each line within a vertically aligned comment block.

3.4 Indents and Tabs

Consistent indenting is part of making your code readable. Four (4) space indents seem to be best (this is also the Microsoft Visual C++ standard). "Soft tabs" (where spaces are used instead of hard tab characters) are recommended to allow the code to be viewed in any editor, and also make it simple to cut and paste code samples into Word, HTML, Help or other files.

Indent the Outer Block Of Each Function

Indent starting at the outer block of each function:

```
void foo( void )
{
    int x;                  /* GOOD */
    if ( x )
        yyy()
    more_code();
    even_more_code();
}
```

Not indenting at the outer block makes it more difficult to locate the top of the function:

```
void foo( void )
{
int x;                          /* POOR */

if ( x )
    yyy()
more_code();
even_more_code();
}
```

Comments And Variables Should Be At The Same Indent Level As the Code

Comments and variable declarations should be indented at the same level as the code:

```
foo()
{
    /* Here we have some comments and
     * declare some local variables.
     */
    int i;

    /* Here we describe what
     * the code does.
     */
    i = code();
    if ( i )
    {
        /* Here are some comments
         * describing what this code does.
         */
        int j;

        j = more_code();
    }
} /* foo() */
```

Indent Statements Associated With Flow-Control Statements

Indent statements associated with flow-control statements `if`, `else`, `for`, `while` or `do`.

```
if ( byLand )
    ShowOne();                                      /* GOOD */
else /* bySea */
    ShowTwo();
```

Conditional sections of flow control should be broken into multiple lines, as illustrated above. Avoid using single line flow control statements. These are not only harder to read, but are difficult debug and breakpoint:

```
if ( byLand ) ShowOne() else ShowTwo();           /* POOR */
```

An exception to this might be if the code can be formatted into neat columns:

```
if      ( byLand ) ShowOne();
else if ( bySea  ) ShowTwo();                     /*  OK  */
else   /* byAir */ ShowThree();
```

3.5 Use Braces When There Are Multiple Lines Under A Flow-Control Statement

```
if ( byLand )
{                                               /* GOOD */
    /* This is a good rule even when the
     * additional lines are only comments.
     */
    ShowOne();
}
```

Without braces it's easy to insert an additional statement that breaks the code:

```
if ( byLand )
    /* Here the code breaks when we just insert the
     * LightLamp() statement and miss the proper bracing.
     */
    LightLamp();
    ShowOne();                                  /* OUCH! */
```

Vertically Align Matching Braces

Finding a missing brace can be a problem when code blocks become especially long. Vertically aligning braces at the outer level makes it much easier to see how they're matched:

```
if ( some_condition )
{
    /* inner block */
}
```

K&R style bracing makes it much more difficult to match up brace pairs:

```
if ( some_condition ) {             /* not recommended */
    code();
}
else {
    more_code();
}
```

Again, vertically aligning brace pairs makes it easier to spot missing braces, and visually easier to locate the inner code blocks:

```
if ( some_condition )
{                                   /*   preferred   */
    code();
}
else
{
    more_code();
}
```

3.6 Add A Closing Comment At The End Of Heavily Nested Code Blocks

Use a closing comment to mark the ending scope for long or heavily nested compound statements:

```
while ( a < b )
{
    while ( something_else() )
    {
        for ( i=10; --i>0; )
        {
            for ( j=10; --j>0; )
            {
                // many lines of code here
```

Down later:

```
            } /* for ( j=10; --j>0; ) */

            // some more lines of code
        } /* for ( i=10; --i>0; ) */

        // maybe some more lines of code
    } /* while ( something_else() ) */

    // maybe some more lines of code
} /* while ( a < b ) */
```

Make sure the comment clarifies the close of the associated statement. The following closing comments are too terse to be useful:

```
            } /* for */

            // maybe some more lines of code
        } /* for */

        // maybe some more lines of code
    } /* while */

    // maybe some more lines of code
} /* while */
```

You can skip the closing comment when the statements are short and the nesting is clear:

```
while ( a < b )
{
    for ( i=10; --i>0; )
    {
        f( i );
    }
}
```

3.7 Add A Closing Comment At The End Of Each Function

A closing comment on the closing brace of a long function makes it easier to locate the function when scanning backwards through the file.

```
void foo( void )
{
    lots_of_code();

} /* foo() */
```

3.8 Add Identifying Comments At The Beginning and End Of Each File

When you have multiple files open for editing, including an identifying comment at the beginning and end of each file helps confirm which file you're working on. A closing comment at the end of a file is especially helpful in verifying that the file is intact and has not been inadvertently truncated at some point.

```
/* foo.c - Copyright (c) 1999  Nobody better use the function name foo again */

void foo( void )
{
    lots_of_code();

} /* foo() */

/* End of File - foo.c */
```

Additional C and C++ code templates are illustrated in Appendix D-G.

3.9 Neat Columns Are Easier To Read

Formatting and making your code easy to read is part of good documentation. Organizing your code into a "tabular" format can help in this regard. Variable declarations, for example, can be formatted into "type", "name" and "description" columns. Notice that while the following two code blocks are the functionally the same, the second one is visually much easier to read.

```
/* Tightly grouped, non-columnized code is hard to read. */

int x; /* description of what X is */
ULONG (*foo)(); /* description of what foo does */
int *iPtr; /* description of what iPtr is */
int z;  /* description of what z is */

x=10; /* related comment */
iPtr = &x; /* another comment */
z = *iPtr + 250; /* and another */
```

compared to:

```
/* With some added white space and columns
 * the code is visually much easier to read.
 */
int     x;          /* description of what X is     */
ULONG (*foo)();     /* description of what foo does */
int    *iPtr;       /* description of what iPtr is  */
int     z;          /* description of what z is     */

x    = 10;          /* related comment */
iPtr = &x;          /* another comment */
z    = *iPtr + 250; /* and another     */
```

3.10 Keep Functions and Parameter Lists Together

An argument list is an integral part of a function call and should not be separated from the function name. The left parenthesis should start immediately following the function name (no space), followed by the list of arguments (each preceded by a space), and then the closing right parenthesis (preceded by a space).

```
foo( arg1, arg2, arg3 );      /* good */

foo (arg1,arg2,arg2);         /* poor */
```

3.11 `if` Is Not A Function Call

An `if` statement should be written to read as a simple sentence. To make it easier to read leave a space after `if`, `else if` or `else` and the associated conditional expression. Also leave some white space after the opening left parenthesis, and before the closing right.

```
if ( todayIsThursday || todayIsFriday )
{
    MakeWeekendPlans();
}
```

3.12 When Declaring Pointers, Use `*` Preceding The Symbol Name

When declaring pointers, place the `*` immediately prefixing the symbol name - not as a suffix to the type.

```
int* x,y;   /* poor – both x & y appear to be pointers – but y isn't!   */
int *x,y;   /* good – here it's clearer that x is a pointer and y isn't */
```

3.13 Minimize Creating Unnecessary Types

For virtually all basic types, Microsoft also defines associated pointer types such as INT_PTR, LONG_PTR, ULONG_PTR, etc. Not only does this clutter the name space, but anytime someone reading the code sees one of these types they have to ask themselves "how is this type defined and why is it so special?".

`INT`, `LONG` and `ULONG` are defined to facilitate changes for machine dependency. INT_PTR, LONG_PTR and ULONG_PTR simply add more details to track. One detail by itself is not a problem, but take on an application with hundreds of minor details and you just add to your support overhead. Keep things simple, using `INT *`, `LONG *` and `ULONG *` makes things explicitly clear to even a work study support programmer reading your code.

4 General Programming

4.1 Avoid Placing Assignment Statements (=) in Conditional Expressions

While close to the heart of many programmers (yes, I admit it), avoid placing an assignment within a conditional expression.

```
if (  ( c=getchar() ) != EOF  )     /* poor      */

c = getchar();
if ( c != EOF )                     /* preferred */
```

If you place an assignment within any conditional, then you must turn off the compiler error detection for "assignment in conditional expression" warnings. This prevents you from detecting common mistypes such as

`if ( x = 2 )` where you meant to say `if ( x == 2)`.

4.2 Put the Shortest Part of an `if/else` On Top

Frequently an `if/else` will have one short error handling section, and one long code block that does most of the actual work. Your code will be more readable if you start with the shortest clause near the top.

```
if ( errorCondition )
    HandleError();
else
{
    /* many lines of code here */
}
```

4.3 Keep Only Loop Control Items in the `for` Statement

The purpose of a `for` statement is to organize the initialization, test and increment part of the loop control into one place. Don't clutter it up with things that have nothing to do with the loop control. Try to avoid shortcuts such as the following:

```
void foo()
{
    enum { arraySize=50 };
    int  array[arraySize];
    int  i;
    int *ptr;

    for ( ptr = array, i = ARRAY_SIZE; --i  >= 0; f(ptr++) )
        ;
}
```

Keep only the loop control variables in the `for` statement to keep things clear:

```
void foo()
{
    enum { arraySize=50 };
    int  array[arraySize];
    int  i;
    int *ptr;

    ptr = array;
    for ( i = arraySize; --i  >= 0; )
        f(ptr++);
}
```

Avoid Declaring Variables Within Loop Control Statements

```
void foo()
{
    enum { arraySize=50 };
    int  array[arraySize];
    int *ptr;

    lots_of_code_here()
    ptr = array;
    for ( int i = arraySize; --i  >= 0; )
    {
        lots_of_more_code_here()
    }
}
```

In the sample above, it appears that `i` is only in scope within the `for` loop. In fact, the scope of `i` is at the next outer level. If we wish to add another `for` loop with `i` as a loop counter later, you'll get a compiler error if you incorrectly redeclare `i`:

```
void foo()
{
    enum { arraySize=50 };
    int  array[arraySize];
    int *ptr;

    lots_of_code_here();
    ptr = array;
    for ( int i = arraySize; --i  >= 0; )
    {
        lots_more_code_here();
    }

    bunches_of_more_code_here();

    for ( int i = 10; --i  >= 0; )  /* error - the previous 'for' has already
    {                                * declared i in the outer scope.
        again_more_code_();          */
    }
}
```

If there's a lot of code, finding where `i` is declared and seeing it's scope is difficult when if it's imbedded within a control statement.

Declare Variables At the Top Of Their Scope

As illustrated in the above code sample, trying to locate variable declarations can become difficult if they're inserted randomly within large sections of code. Defining variables at the top of their scope makes it easier to locate their declarations:

```
void foo()
{
    enum { arraySize=50 };
    int  array[arraySize];
    int  i;               /* here it's easy to locate the declaration for i */
    int *ptr;

    lots_of_code_here();
    ptr = array;
    for ( i = arraySize; --i  >= 0; )
    {
        int j = i;          /* If we need, we can declare variables
        int k = 0;           * within the scope of the for loop here.
                             */
        lots_ more_code();
    }
}
```

4.4 Use Debug ASSERT's Liberally

`_ASSERT()` (standard C) and `ASSERT()` (C++) macros provide a simple mechanism for checking assumptions during debugging – use them liberally! Debug `_ASSERT()` and `ASSERT()` statements are only compiled when the `_DEBUG` identifier is defined. When `_DEBUG` is not defined, `_ASSERT()` and `ASSERT()` statements are removed by the precompiler and perform no operation.

Assert statements perform a test for a specified condition, and if the test condition is not met cause a debug breakpoint to occur. When used with `MRESULT` function return codes (below), assert statements can also be used to distinguish between "normal completion" and "non-fatal/warning completion" cases during debug:

Assert debug breakpoint for either "fatal error" or "non-fatal/warning" return conditions (no breakpoint on "normal completion"):

```
MRESULT result;
```

```
result = mgReadHeader();
_ASSERT( WARNINGFREE(result) );  /* debug breakpoint if result!=0 */
if ( FAILED(result) )      /* check for result<0 fatal error case */
{
    /* handle fatal error case */
}
```

Assert debug breakpoint for "fatal error" return conditions only:

```
MRESULT result;

result = mgReadHeader();
_ASSERT( SUCCEEDED(result) ); /* debug breakpoint if result<0        */
if ( FAILED(result) )         /* check for result<0 fatal error case */
{
    /* handle fatal error case */
}
```

`_ASSERT()`, `SUCCEEDED()`, `FAILED()` and `WARNINGFREE()` are macros defined in the Metagraphics `METINCS.H` header file (see Appendix B):

4.5 Check Function Return Codes

Functions and methods should be written to return a failure, success or warning return code (this also helps structure your code for use with COM, if desired later). Where possible, return an `MRESULT` code. `MRESULT` is a signed `long` integer value whose return code indicates the following basic conditions:

<0	Failure	Negative value contains an fatal error return code along with a library and function ID.
=0	Success	Normal completion
>0	Success	Positive value contains a special non-fatal/warning completion code along with a library and function ID.

`MRESULT's` along with the `_ASSERT()`, `ASSERT()`, `SUCCEEDED()`, `FAILED()` and `WARNINGFREE()` macros can greatly aid in both documenting and debugging your code:

```
MRESULT result;

result = mgReadHeader();
_ASSERT( WARNINGFREE(result) );  /* debug breakpoint if result!=0 */
if ( FAILED(result) )      /* check for result<0 fatal error case */
{
    /* handle failure situation */
}
```

4.6 Always Have a Default Case With `switch` and `if/else if` Statements

A `switch` statement should always have a `default:` case, especially if the default shouldn't happen. If the default case is illegal, log it, return an error code or do something graceful.

```
switch ( i )
{
    case 1:  DoSomething();   break;
    case 2:  DoOtherthing();  break;
    default:
    {
        _ASSERT( FALSE );  /* debug trap */
        ErrorMessage( "illegal value for i" );
    }
}
```

The same guideline applies when using `if` with `else if` statements. Always include a default `else` statement at the end to trap unexpected conditions.

```
if ( i == 1)
    DoSomething();
else if ( i==2 )
    DoOtherthing()
else /* handle unexpected condition */
{
    _ASSERT( FALSE );  /* debug trap */
    ErrorMessage( "illegal value for i" );
}
```

4.7 Don't Divide By Zero

Precede all division statements for an explicit check for divide by zero. If the dividend is zero, log it, return an error code or do something graceful (don't throw a "divide by zero" exception).

4.8 Initialize All Pointers

Initialize all pointers when they are declared with some valid value. If no value is available when declared, initialize the pointer to `NULL`.

4.9 Enhancing `struct` Compatibility

Even with the best design planning, it's often necessary to add new variables to existing data structures to support new features or capabilities in your program. Data structure extensibility becomes an issue, however, unless all the components for creating and accessing the data are kept exactly in sync. Struct "versioning" problems may occur in many common situations:

1. The data structure is written as part of a file that is stored on disk and read back later for processing (potentially by a new version of the program).
2. The structure is used as part of a data transfer for information electronically to another machine (potentially by a different version of the program).
3. Portions of the application are compiled and linked at different times (such as DLL's and COM modules).

Whenever components of an application, utility or shared library are built as separate elements, the risk for data structure version incompatibility arises. The following guidelines are designed to enhance structures for extensibility and upward compatibility.

Include a `structSize` Member In Structures

Often with functions that are built into dynamically linked libraries (DLL's) it may be necessary later to update a function which, in turn, may require adding additional items to a structure that is passed by pointer reference. Version problems will occur if an existing application passes a pointer of an old structure type to a function in a new DLL with an updated structure type.

To simplify updates and to facilitate DLL compatibility between versions, *unless you are absolutely certain that a structure will never change*, include an `INT32 structSize` variable as the first data member of all structures. By examining the `structSize` variable, updated functions can determine specifically which version of a structure is being passed, and then act accordingly in a compatible manner. The `_InitStruct()` macro is a convenient method to initialize a structure to zero and then automatically set the `structSize` member.

The application code would be something like this:

```
typedef struct _fooStruct
{
    INT32   structSize;  /* size of this structure */
```

```
        /* other structure members defined here */
    } fooStruct;

        _InitStruct( &myStruct );  /* zero the structure and set structSize */
        /* set other structure members */

        foo( &myStruct );   /* call a function passing a pointer to the struct */
```

The called function code can now distinguish between different structure versions:

```
    typedef struct _FOOSTRUCT_REV1  /* old version of FOOSTRUCT */
    {
        INT32   structSize;  /* size of this structure */
        /* other structure members defined here */
    } FOOSTRUCT_REV1;

    typedef struct _FOOSTRUCT   /* current version of FOOSTRUCT */
    {
        INT32   structSize;  /* size of this structure */
        /* original structure members defined here */
        /* new structure members defined here */
    } FOOSTRUCT;

    void foo( FOOSTRUCT *fooStruct )
    {
        if ( fooStruct->structSize == sizeof(FOOSTRUCT_REV1) )
        {
            /* cast fooStruct to FOOSTRUCT_REV1 and process accordingly */
        }
        else if ( fooStruct->structSize == sizeof(FOOSTRUCT) )
        {
            /* process fooStruct with current FOOSTRUCT definition */
        }
        else
        {   /* We can also double check we are getting passed a valid parameter. */
            /* Something is bad if FOOSTRUCT doesn't match any size we expect!   */
            _ASSERT( FALSE );  /* debug trap */
        }
    } /* foo() */
```

A `structSize` variable automates the older technique of having a manually updated version number at the beginning of a data structure.

Include a `structSize` Parameter For Functions Returning Structure Data

With a `structSize` variable as the first element in a structure, when a function is called passing the structure as input the function can now easily identify a specific structure version (see above). Returning data in a structure passed by pointer reference also creates potential versioning problems. If only a pointer to an empty structure is passed, the called function can only assume its size and version. If the caller's structure version is out of date and too small, data overwrite errors can easily occur.

To enhance compatibility for structures passed by pointer for returning data, include a `structSize` parameter along with the structure pointer parameter passed to the function. The `structSize` parameter allows the called function to identify specifically which version of a structure is being passed, and then act accordingly in a compatible manner.

```
    FOO         myFoo;       /* FOO object instance handle */
    FOOINFO     myFooInfo;   /* FOOINFO data structure     */
    MRESULT     result;      /* function return code       */

    /* create a FOO instance (foo instance handle is returned) */
    result = Foo_Create( &myFoo );
    _ASSERT( SUCCEEDED(result) );
```

```
/* get FOOINFO struct data */
result = Foo_GetFooInfo( myFoo, sizeof(FOOINFO), &myFooInfo );
_ASSERT( SUCCEEDED(result) );
```

By explicitly passing the size of the structure, the called function can now distinguish between different structure versions:

```
/* Return FOOINFO structure data */
MRESULT Foo_GetFooInfo(
    FOO       foo,           /* input,  foo handle              */
    int       fooInfoSize,   /* input,  fooInfo struct size     */
    FOOINFO  *fooInfo );     /* output, fooInfo information     */
{
    if ( fooInfoSize == sizeof(FOOINFO_REV1) )
    {
        /* cast fooInfo to old FOOINFO_REV1 and process accordingly */
    }
    else if ( fooInfoSize == sizeof(FOOINFO) )
    {
        /* process fooInfo with current FOOINFO definition */
    }
    else
    {   /* We can also double check we are getting passed a valid parameter. */
        /* Something is bad if fooInfoSize doesn't match any size we expect! */
        _ASSERT( FALSE );  /* debug trap */
    }
} /* Foo_GetFooInfo() */
```

4.10 Avoid Magic Numbers

Except for possibly 0 or 1, the main body of your code should avoid using explicit numbers. Use `enum` or `const` to give a number a symbolic name. This provides two advantages:

- The symbolic name helps document what the value is and what it's used for.
- If the number is used in more than one place, there's only one spot to change.

Locally used variables can be an exception to this:

```
foo( TCHAR *fullname )
{
    TCHAR firstname[256], lastname[256];  /* this is ok */
    ...
    /* get the individual firstname and lastname strings */
    GetNames( fullname,
              firstname, _ArrayCount(firstname),
              lastname,  _ArrayCount( lastname) );
}
```

Because the `_ArrayCount()` macro is used, the number of elements in each array is automatically supplied (and works both for ASCII or Unicode!).
C and C++ Portability Rules

5 Code Optimization

The following items relate to general techniques for code optimization. While some of these optimizations are specific to Intel processors, even these are good points to check when working on other machines.

5.1 Count Down to Zero in `for` Loops

Testing against zero is usually much more efficient than repeatedly testing for an explicit value. With a good compiler, a pre-decrement will also set the condition code so that a separate test for zero is not even necessary. Instead of:

```
for ( i=0; i < max; i++ )
{
    /* loop code here */
}
```

use:

```
for ( i=max-1; --i >= 0; )     /* faster  */
{
    /* loop code here */
}
```

While it's generally preferable to keep the loop control variables together at the beginning at the loop, for extremely time-critical loops replace the above `for` loop with a `do/while` loop to streamline the assembly language code for faster execution:

```
i = max - 1;
do                             /* fastest */
{
    /* loop code here */
} while (--i >= 0 );
```

Pre-Increment/Decrement Is Faster Than Post

For Intel processors, using predecrement or preincrement operators (e.g. `--i` or `++i`) can save a memory reference and are generally faster than using post-decrement/post-increment operators (e.g. `i--` or `i++`).

5.2 Unrolling Loops

Unrolling loops is a simple way to speed performance. This works best if you can keep the list of loop instructions small enough to still fit within the processor instruction cache. For Pentium processors, L1 cache is 128 bytes.

Typical Loop:

```
checksum = 0;
for ( n=0;  n<32768; ++n )
    checksum += array[n];
```

Unrolling the loop improves the speed 50% or more:

```
checksum = 0;
for ( n=0;  n<32768; n+=8 )
{
    checksum += array[n+0];
    checksum += array[n+1];
    checksum += array[n+2];
    checksum += array[n+3];
    checksum += array[n+4];
    checksum += array[n+5];
    checksum += array[n+6];
    checksum += array[n+7];
}
```

Also for Intel processors, accessing arrays using indexes can be more than two times faster than using incremented pointers (the compiler uses the new `*[ptr+const]` instruction address mode).

5.3 Keep Cache Usage In Mind

Keep usage of the processor cache in mind – *this can significantly speed up your code!* L1 is the primary on-chip cache, and is either 8Kb or 16Kb in size. Access is organized in 32-byte groups. For fastest speed, process blocks of data in 16Kb (or less) groups, and try to align important addresses on 16 byte boundaries.

5.4 Data Layout Is Important

Improper data structure layout can have a detrimental effect on program speed. Keep in mind the following guidelines:

- Don't pack data structures. This can cause data to be unevenly aligned and cause memory access to be slower. Misalignment of data is expensive – it can cost up to 12 clocks per access!
- Pay careful attention to the size and alignment of structures and arrays. Keep the cache in mind when creating structures.
- Try to put frequently accessed data next to each other to keep it in the cache.
- Align 16-bit values on 2 byte boundaries.
- Align 32-bit values on 4 byte boundaries.
- Align 64-bit values on 8 byte boundaries.
- Align data structures and arrays greater than 32 bytes on 32 byte boundaries. If you have a large structure, make *sure* that it gets allocated on a 32-byte boundary (this may mean writing your own memory allocator).
- Most compilers will do the right thing for all data types except `char`.
- Put data members in a struct in size order, largest first.

5.5 Profile Your Code

Use a profiler to identify the hot spots in your code (chances are they're not where you think). The Microsoft *Visual C++ Profiler* is ok; Intel *VTune* or NuMega *TrueTime* are better. Once you identify a hot spot, before tweaking the code, think through if there might be a faster or better way to perform the same operation.

Turn Off Incremental Linking

Incremental linking bloats the code size, and also skews profiling and performance results.

6 Portability Guidelines

The following set of guidelines are useful insuring that your code is portable to a broad range of C/C++ compilers and OS platforms. Most of these are related to relatively new C/C++ language features that are not yet fully or consistently implemented across various compilers. For all of these situations there are usually simple workarounds that operate more reliably and that are supported by all compilers.

6.1 Avoid Using Templates

Don't use the C++ template feature. This feature is still not implemented by all compilers, and even when it is implemented, there is great variation. Most of the interesting things that you would want to do with templates (type safe container classes, etc.) can be implemented with macros and casting, even though you do lose the type safety (pity). Often times subclassing can easily achieve the same result. Templates also slow compilation, 200%-300% is not uncommon, and their support in many compilers that offer it is still subject to errors (gcc is such an example).

<u>Workaround:</u> The things you would like to use templates for are, most commonly, polymorphic containers (in the sense that they can contain objects of any type without compromising C++ type system, i.e. using void * is out of question). Lack of templates is not a reason to use static arrays or typeless (passing by `void *`) containers.

6.2 Don't Using Exceptions

C++ exceptions are another feature that are not widely implemented, and as such, their use makes your code compiler and platform specific.

Workaround: There is no real workaround, of course, or the exceptions wouldn't have been added to the language. However, here are a few suggestions that might help:

- Every function should returns an integer (or at least boolean) error code.
  There is no such thing as a function that never fails - even if it can't fail now, it will likely later, when modified to be more powerful or general. Put an integer return type from the very beginning (see Appendix B, MRESULT)!

- Every function you call may fail - check the return code!
  Never rely assuming a function's success, always test for a possible error.

- Tell the user about the error, don't silently ignore them.
  Exceptions are always caught and, normally, processed when they're caught. In the same manner, the error return code must always be processed somehow. You may choose to ignore it, but at least log the error and tell the user that something wrong happened.

One exception to this rule (don't say it) is that it's probably ok, and may be necessary to use exceptions in some machine specific code. If you do use exceptions in machine specific code you must catch all exceptions there because you can't throw the exception across cross platform code.

6.3 Don't Use Runtime Type Information (RTTI)

Run-time type information (RTTI) is a relatively new C++ feature, and not supported in many compilers.

Workaround: If you need runtime typing, you can achieve a similar result by adding a `classOf()` virtual member function to the base class of your hierarchy and overriding that member function in each subclass. If `classOf()` returns a unique value for each class in the hierarchy, you'll be able to do type comparisons at runtime.

6.4 Don't Use Namespace Facility

Support of namespaces (through the `namespace` and `using` keywords) is a relatively new C++ feature, and not supported in many compilers.

6.5 Don't Put C++ Comments in C Code and Headers

Never use C++ comments in C code - not all C compilers/preprocessors understand them. (Yes, this works for Microsoft Visual C/C++, but it is not supported by many of compilers – don't go there.)

Many header files will also be included both by C files and C++ files. Use this same rule with header files. Don't put any C++ "`//`" style comments in .h header files that might be included by C files. You might argue that you could use C++ style comments inside `#ifdef __cplusplus` blocks, but these will not work in all cases (some compilers have weird interactions between comment stripping and pre-processing). It's not worth the effort and risk. Stick to C style `/**/` comments only for any header file that may ever likely to be included by a C file. To keep things simple:

- Use "`//`" style comments only in C++ .cpp and .hpp files
  (.hpp files may only be included in .cpp and other .hpp files) .

- Use "`/* */`" comments in C .c and .h files (also .cpp and .hpp if you wish)
  (.h files may be included from either .cpp, .hpp, .c or other .h files).

6.6 Make C Header Files Compatible with C and C++

Make the header files work correctly when included by both C and C++ files. If you include an existing C header in new C++ files, fix the C header file to work properly with both C and C++.

Poor – don't just `extern "C" {}` old header files in your .cpp program:

```
/* oldCheader.h */
int existingCfunction( char * );
int anotherExistingCfunction( char * );

/* oldCfile.c */
#include "oldCheader.h"
...

// new file.cpp
extern "C"              // poor - don't just extern C in your .cpp file
{
#include "oldCheader.h"
};
...
```

Correct – Update C header files to work properly both C and C++:

```
/* oldCheader.h */
#ifdef __cplusplus
extern "C" {            /* correct - insert extern C in the .h file to make */
#endif                  /* the header directly compatible with both C & C++ */
int existingCfunction( char * );
int anotherExistingCfunction( char * );
#ifdef __cplusplus
}
#endif

/* oldCfile.c */
#include "oldCheader.h"
...

// new file.cpp
#include "oldCheader.h"
...
```

6.7 Manually Initialize Automatic Array Variables

Initialization of non-static automatic array variables is not supported in some compilers. For example:

```
void FuncFoo()                              /* non-portable!!! */
{ /* some compilers can not initialize automatic array variables */
    int myArray[] = { 1, 2, 3 };
}
```

will fail to compile with HP-UX C++ compiler. Only use array initializers with `static` or `static const` arrays. If you meant the array to be `static` or `static const`, specify it as such. Otherwise, you should manually initialize each array variable by hand:

```
void FuncFoo()                              /* portable        */
{/* initialized once - changes carry over between multiple calls */
    static int myArray[] = { 1, 2, 3 };
      :
}
```

or,

```
void FuncFoo()                              /* portable        */
{/* initialized once - const elements can not be changed        */
    static const int myArray[] = { 1, 2, 3 };
      :
    }
```
or,
```
    void FuncFoo()                              /* portable         */
    {/* manually initialize array variables for every call          */
        int myArray[3];

        myArray[0]=1;
        myArray[1]=2;
        myArray[2]=3;
          :
    }
```

7 Windows Programming

7.1 SourceSafe Files

For Windows C and C++ projects, the following files should be checked into source control:

Check In:

`.dsw` – Workspace file
`.dsp` – Project file(s)
`*.cpp, *.c, *.hpp, *.h, *.rc, resource.h` – Source files
`.obj` or `.lib` – any that aren't built within the project
`.doc` – Documentation files

Don't Check In:

`.obj` – project compiled object modules
`.idb` – partial debugger info (used by linker)
`.pdb` – debug info
`.opt` – options file (window size/position, menu views, doc views, etc.)
`.plg` – IDE build log
`.ncb` – no-compile browser (ClassView)
`.pch` – pre-compiled headers
`.ilk` – incremental linker temp file

7.2 Visual C++ #pragma's

Compile at Maximum Warning Level 4

To the maximum extent possible, write your code to compile both totally error and warning free. In this regard, compile at the highest warning level possible, Warning Level 4, to allow the compiler to perform the maximum amount of error checking. Unfortunately, Microsoft's Operating Systems Group doesn't share these sentiments about Warning Level 4, and many of the Windows header files and Windows macros generate extraneous warning messages due to unconventional coding techniques. The following "`#pragma`" statements disable the compiler from reporting certain Level 4 Warnings commonly generated from the Windows header files and macros. (Note - pragma's that are `//` commented out below were reported as problems by Jeffrey Richter in his book "*Advanced Windows NT*". We have not seen these warnings from Windows code as yet, and have left them as comments for the time being.)

```
    /* nonstandard extension 'single line comment' was used        */
    #pragma warning(disable: 4001)

    /* indirection to slightly different base types                */
    //#pragma warning(disable: 4057)
```

```
/* unreferenced formal parameter                            */
#pragma warning(disable: 4100)

/* named type definition in parentheses                     */
//#pragma warning(disable: 4115)

/* conditional expression is constant (_ASSERT macro)       */
#pragma warning(disable: 4127)

/* nonstandard extension used : nameless struct/union       */
/* (winnt.h,winbase.h,mmsystem.h)                           */
#pragma warning(disable: 4201)

/* nonstandard extension used : benign typedef redefinition    */
//#pragma warning(disable: 4209)

/* nonstandard extension used : bit field types other than int */
//#pragma warning(disable: 4214)

/* unreferenced inline function has been removed            */
#pragma warning(disable: 4514)

/* Note: Creating precompiled header                        */
//#pragma warning(disable: 4699)
```

You can also use `#pragma warning(push)` and `#pragma warning(pop)` to save and restore the current warning state (see "#pragma" documentation on the MSDN Library CD for more information).

Specify Default Libraries

Use a `#pragma comment(lib,"libname")` statement in your header files to specify which default libraries are needed when linking your program. For Visual C++, this `#pragma` inserts a special comment into the .obj files that tells the linker which default libraries are required. Rather than generating a list of ambiguous "unresolved external" errors, the linker will automatically link the library, or it will print an error message with the missing library name if it can't be found in one of the default paths.

```
#pragma comment(lib,"comctl32.lib")    /* common controls lib must be linked */
```

Compile-Time Messages

You can mark a location in your code with a compile-time message using the `#pragma message(` *`messagestring`* `)` statement. The message shows is displayed in the compiler display window, but does not effect the compile. This is handy when you need to leave yourself a reminder of a code area that you need to come back to. `#pragma` messages can also include a reference to the specific filename and line number:

```
#pragma message( "Don't forget to come back and test the code logic here" )
#pragma message( "Additional code logic needed here -" __FILE__ __LINE__)
```

7.3 Windows Timers and Timing

Basic Timers – `SetTimer()` and `KillTimer()`

`SetTimer()` and `KillTimer()` set up a timer to send a WM_TIMER message or to invoke a callback function. These functions are easy to use, but the timing resolution of the messages coming back is not very accurate. SetTimer() messages may be delayed and/or concatenated (interim messages lost). These timers provide a general resolution of 20ms under Windows 95, and 5ms under Windows 98. These functions should not be used time-critical operations.

Multimedia Timer – `timeGetTime()`

The Windows multimedia function `timeGetTime()` returns the time from system startup in milliseconds (wraps about every 50 days). This timer has a resolution of ~1ms for Windows 98, and ~5ms for Windows NT4 (under NT the resolution can also be changed).

High Performance Timer – `QueryPerformanceCounter()`

The new Win32 high performance timing functions, `QueryPerformanceCounter()` and `QueryPerformanceFrequency()`, provide high performance and high resolution timing accurate to 0.8ms or better. The one downside of these functions is that they are only supported when running on a Pentium processor or better (these functions use the new Pentium RDTSC instruction). To operate on other machines you need to code default logic using `timeGetTime()` if you're running on a non-Pentium machine and QueryPerformanceCounter() isn't supported.

```
INT64        performanceFreq;
static int   msPerPerformanceCount
static BOOL  hiResTimerSupported;

hiResTimerSupported = QueryPerformanceFrequency( &performanceFreq );
if ( hiResTimerSupported )
{   /* QueryPerformanceCounter() supported */
    msPerPerformanceCount = 1000.0F/performanceFreq;
}
else
{   /* QueryPerformanceCounter() not supported, must use timeGetTime() instead */
    if ( _WINNT )
        msPerPerformanceCount = 5;
    else /*_WIN98*/
        msPerPerformanceCount = 1;
}
```

Here's an outline of a sample piece of code to calculate the number frames per second:

```
void Render()
{
static INT64 startCount, stopCount, previousCount;
int  framesPerSecond;

if ( hiResTimerSupported )
    QueryPerformanceCounter( &startCount );
else
    startCount = timeGetTime();

/* a bunch of code here to render the scene */

previousCount = stopCount;
if ( hiResTimerSupported )
    QueryPerformanceCounter( &stopCount );
else
    stopCount = timeGetTime();

msTimeToRender  = msPerPerformanceCount * (stopCount - startCount);
framesPerSecond = 1.0f /
           ( 1000.0*msPerPerformanceCount * (stopCount-previousCount) );
}
```

7.4 Drawing With GDI and DirectX

GDI and DirectX (and its components DirectDraw, Direct3D, DirectSound) are the major display drawing interfaces when working under Windows. GDI (Graphics Device Interface) is the original and most commonly interface for drawing to display devices.

Why use GDI?

- Because it’s there.
- It’s well understood.
- It offers lots of functionality (text, 2D stuff)
- It works.
- It’s ok for non-realtime rendering.

Why Not Use GDI?

- It was written by 1000 monkeys.
- It’s incredibly slow.
- It’s very cumbersome needing to continually rely on DC’s

For Fastest Performance Use DirectX

For real-time rendering, DirectX offers several major performance advantages:

- Provides nearly direct access to the hardware
- Allows you to configure the display however you like.
- You can mix GDI calls with DirectX.
- You get access to 2D/3D hardware acceleration, true alpha blending, 3D and video! (every new PC shipping today contains a reasonably good 2D/3D accelerator chip)

Appendix A - Computational Data Types

An additional set of computational data types are frequently used for graphic and display drawing operations. These computational data types include:

Fixed-Point Data Types (defined in mgtypes.h)

Data Type	Win32 Type	Description
F2DOT14	`short`	Signed 2.14 fixed-point fractional integer (16-bits)
F12DOT4	`short`	Signed 12.4 fixed-point fractional integer (16-bits)
F26DOT6	`long`	Signed 26.6 fixed-point fractional integer (32-bit)
⬄ **FIXDOT**	`int`	`F12DOT4` or `F26DOT6` depending if `int` is 16- or 32-bits.

⬄ Indicates machine-dependent variable size data type.

The `F2DOT14` ("fixed-point 2.14") data type consists of a signed 2-bit mantissa and a positive unsigned 14-bit fraction (16-bits total). This data type is used for unary vectors, sine and cosine values, and scaling coefficients. To compute the decimal value, add the positive fraction to the signed mantissa. Examples of `F2DOT14` values are:

Decimal Value	Hex Value	Mantissa	Fraction
0.0	0x0000	0	+0/16384
0.000061035	0x0001	0	+1/16384
0.25	0x1000	0	+4096/16384
0.5	0x2000	0	+8192/16384
1.0	0x4000	1	+0/16384
1.75	0x7000	1	+12288/16384
-1.0	0xC000	-1	+0/16384
-0.5	0xE000	-1	+8192/16384
-2.0	0x8000	-2	+0/16384
+1.999938964	0x7FFF	+1	+16383/16384

The `F12DOT4` (fixed-point 12.4) data type consists of a 12-bit signed mantissa and a positive unsigned 4-bit fraction (16-bits total). This data type is typically used to define fractional pixel positions on 16- and 8-bit processors. To compute the decimal value, add the positive fraction to the signed mantissa. Examples of `F12DOT4` values are:

Decimal Value	Hex Value	Mantissa	Fraction
0.0	0x0000	0	+0/16
0.0625	0x0001	0	+1/16
0.25	0x0004	0	+4/16
0.5	0x0008	0	+8/16
1.0	0x0010	1	+0/16
1.75	0x001C	1	+12/16
-1.0	0xFFF0	-1	+0/16
-0.5	0xFFF8	-1	+8/16
-2048.0	0x8000	-2048	+0/16
+2047.9375	0x7FFF	+2047	+15/16

The `F26DOT6` (fixed 26.6) data type consists of a 26-bit signed mantissa and a positive unsigned 6-bit fraction (32-bits total). This data type is typically used to define fractional pixel positions on 32- and 64-bit processors. To compute the decimal value, add the positive fraction to the signed mantissa. Examples of `F26DOT6` values are:

Decimal Value	Hex Value	Mantissa	Fraction
0.0	0x00000000	0	+0/64
0.015625	0x00000001	0	+1/64
0.25	0x00000010	0	+16/64
0.5	0x00000020	0	+32/64
1.0	0x00000040	1	+0/64
1.75	0x00000070	1	+48/64
-1.0	0xFFFFFFC0	-1	+0/64
-0.5	0xFFFFFFE0	-1	+32/64
-33,554,432.0	0x80000000	-32,554,432	+0/64
+33,554,431.984375	0x7FFFFFFF	+33,554,431	+63/64

`FIXDOTX` is a conditional data type defined to either `F12DOT4` or `F26DOT6` dependent on the native size of type `int`. For 16-bit compilers, `FIXDOTX` is defined equivalent to `F12DOT4`. For 32- and 64-bit compilers `FIXDOTX` is defined equivalent to `F26DOT6`. This data type is typically used to define fractional pixel and angular measurements in a portable, computationally efficient format. Note that care must be taken in using the `FIXDOTX` type since the 16-bit `F12DOT4` data type can only hold a maximum value of 2047.9375).

```
#if    sizeof(int) == 2
#define  FIXDOT  F12DOT4

#elif  sizeof(int) >= 4
#define  FIXDOT  F26DOT6

#else
#error "Unsupported 'int' size!"
#endif
```

Fixed-Point Macros (defined in mgtypes.h)

The Metagraphics header file, mgtypes.h, defines a standard set of macros for use with fixed-point data types:

```
/* F12DOT4 fixed-point macros - - - - - - - - - - - - - - - - */

/* convert integer to fixed-point F12.4 */
#define IntToF12(intValue)  (F12DOT4)( intValue << 4 )

/* round fixed-point f12Value to integer */
#define F12ToInt(f12Value)  (INT)( (f12Value + 0x08) >> 4 )

/* truncate fixed-point f12Value to integer */
#define F12TruncToInt(f26Value)  (INT)( (f12Value) >> 4 )

/* floor f12Value to next lowest integer value */
#define F12Floor(f12Value)  (F12DOT4)( (f12Value) & -16 )

/* ceil f12Value to next largest integer value */
#define F12Ceil(f12Value)   (F12DOT4)( ((f12Value)+15) & -16 )

/* F26DOT6 fixed-point macros - - - - - - - - - - - - - - - - */

/* convert integer to fixed-point F26.6 */
#define IntToF26(intValue)  (F26DOT6)( intValue << 6 )

/* round fixed-point f26Value to integer */
#define F26ToInt(f26Value)  (INT)( (f26Value + 0x20) >> 6 )

/* truncate fixed-point f26Value to integer */
#define F26TruncToInt(f26Value)  (INT)( (f26Value) >> 6 )

/* floor f26Value to next lowest integer value */
#define F26Floor(f26Value)  (F26DOT6)( (f26Value) & -64 )

/* ceil f26Value to next largest integer value */
#define F26Ceil(f26Value)   (F26DOT6)( ((f26Value)+63) & -64 )

/* FIXDOTX fixed-point macros - - - - - - - - - - - - - - - - */

#if    sizeof(int) == 2
#define  FIXDOT         F12DOT4
#define  IntToFix       IntToF12
#define  FixToInt       F12ToInt
#define  FixTruncToInt  F12TruncToInt
#define  FixFloor       F12Floor
#define  FixCeil        F12Ceil

#elif  sizeof(int) >= 4
#define  FIXDOT         F26DOT6
#define  IntToFix       IntToF26
#define  FixToInt       F26ToInt
#define  FixTruncToInt  F26TruncToInt
#define  FixFloor       F26Floor
#define  FixCeil        F26Ceil

#else
#error "Unsupported 'int' size!"
#endif
```

Appendix B - MRESULT Function Return Codes

Most Metagraphics functions provide back an "`MRESULT`" completion code when a function returns. The MRESULT return code is either 16- or 32-bits, dependent on the native integer size for the compiler and target platform. `MRESULT` is conditionally defined to `MRESULT16` or `MESULT32` depending on the integer byte size:

```
#define  MRESULT16  signed short
#define  MRESULT32  signed long

#if    sizeof(int) == 2
#define  MRESULT  MRESULT16
#elif  sizeof(int) >= 4
#define  MRESULT  MRESULT32
#else
#error "Unsupported 'int' size!"
#endif
```

MRESULT Coding **(defined in merror.h)**

Non-zero `MRESULT32` return values return an error code, internal tag location, facility or library, and function ID:

```
     3322 2222 2222 1111 1111 11
Bit: 1098 7654 3210 9876 5432 1098 7654 3210  MRESULT32 signed integer
     s--- ---- ---- ---- ---- ---- ---- ----  success(0) or fail(1)
     -i-- ---- ---- ---- ---- ---- ---- ----  informational(0) or warning(1)
     --ee eeee eeee ---- ---- ---- ---- ----  error code (0-1023)
     ---- ---- ---- tttt ---- ---- ---- ----  tag location (0-15)
     ---- ---- ---- ---- gggg gg-- ---- ----  group* (0-63)
     ---- ---- ---- ---- ---- --ff ffff ffff  function (0-1023)
```

Non-zero `MRESULT16` return values return an error code and internal tag location:

```
     1111 11
Bit: 5432 1098 7654 3210  MRESULT16 signed integer
     s--- ---- ---- ----  success(0) or fail(1)
     -i-- ---- ---- ----  informational(0) or warning(1)
     --ee eeee eeee ----  error code (0-1023)
     ---- ---- ---- tttt  tag location (0-15)
```

* The following groups are currently defined:

0 = Application
1 = Metagraphics MetaWINDOW library
2 = Metagraphics Media!Lab library
3 = Metagraphics Media!Key library
4 = Metagraphics TypeServer library

Appendix C - Global Utility Functions and Macros

Metagraphics header files include a series global utility functions and macros to simplify and standardize common operations. Depending on the target platform and optimization, these may either be implemented as `#define` macros, `inline` functions or standard C functions.

General Utility Functions (defined in metincs.h)

_ZeroMemory()

```
/* clear a block of memory to zero */
inline void _ZeroMemory( void *memory, size_t byteCount )
```

_ZeroStruct()

```
/* clear a structure to zero */
#define _ZeroStruct(structure)  _ZeroMemory(structure, sizeof(*(structure)))
```

_InitStruct() – Zero a structure and set the structSize member

```
/* Clear a structure and initialize the first member to the size of the structure */
#define _InitStruct(structure) {                         \
        _ZeroMemory(&(structure), sizeof(structure));  \
        *(int*) &(structure) = sizeof(structure);      \
        }
```

countof() / _ArrayCount()

```
/* count the number of elements in an array */
#define  countof(array)     ( sizeof(array) / sizeof( (array)[0] ) )
#define _ArrayCount(array)  ( sizeof(array) / sizeof( (array)[0] ) )
```

_InRange()

```
/* this macro evaluates TRUE of val is between lo and hi inclusive */
#define _InRange(lo, val, hi)  (((lo) <= (val)) && ((val) <= (hi)))
```

_FourCC()

```
/* pack 4 character codes into an INT32 */
#define _FourCC( ch0, ch1, ch2, ch3 )                                \
       ( (INT32)(BYTE)(ch0) | ( (INT32)(BYTE)(ch1) << 8 ) |          \
       ( (INT32)(BYTE)(ch2) << 16 ) | ( (INT32)(BYTE)(ch3) << 24 ) )
```

_ASSERT() / ASSERT()

C `_ASSERT()` and C++ `ASSERT()` macros test if an expression is TRUE (non-zero) or FALSE (zero). If FALSE, the assert macro displays a message box indicating the file name, line number and the expression that failed. Assert macros are only active in debug builds where "_DEBUG" is defined. In release builds, assert statements are removed by the compiler preprocessor. For this reason, assert statements should not perform any operations or function calls that are needed for normal execution. (`_ASSERT()`/`ASSERT()` are predefined within Windows crtdbg.h.)

```
/* display a message box if an assertion fails in a debug build */
#ifdef   _DEBUG
#define _ASSERT(x)   if ( !(x) ) _ASSERTFAIL( _FILE_, _LINE_, #x )
#else  /*_RELEASE */
#define _ASSERT(x)
#endif /*_DEBUG   */
```

FAILED() / SUCCEEDED() / WARNINGFREE()

`FAILED()`, `SUCCEEDED()` and `WARNINGFREE()` macros should be used to make status testing `if`, `ASSERT` and `_ASSERT` statements more readable.

```
/* Test for result failure.  Negative values indicate failure.            */
#define FAILED(status)      ( (MRESULT)(status) <  0 )

/* Test for result success.  Non-negative values indicate success.        */
#define SUCCEEDED(status)   ( (MRESULT)(status) >= 0 )

/* Test for result success.  Zero value indicates error and warning free. */
#define WARNINGFREE(status) ( (MRESULT)(status) == 0 )

/* Test for result success.  Non-zero value indicates error or warning.   */
#define NOTWARNINGFREE(status) ( (MRESULT)(status) != 0 )
```

Windows Utility Functions

_MessageBox() – Quick Windows MessageBox() macro

```
// quick MessageBox() macro
#define _MessageBox(str) {                                \
        TCHAR szTMP[256];                                 \
        GetModuleFileName(NULL, szTMP, mARRAYCNT(szTMP)); \
        MessageBox(GetActiveWindow(), str, szTMP, MB_OK); \
        }
```

_MSG – Windows #pragma message() macro

```
// _MSG - #pragma message() reminder macro (for compile-time messages)
// When the compiler sees a line like this:   #pragma _MSG(Fix this later)
// it will output a line like this:    C:\medialab\src\xxx.h(304):Fix this later
#define _STR(x)      #x
#define _STR2(x)     _STR(x)
#define _MSG(desc)  message(__FILE__ "(" _STR2(__LINE__) "):" #desc)
```

_HANDLE_DLGMSG() – Message cracker handler for dialog box messages

```
// The normal HANDLE_MSG macro in WINDOWSX.H does not work properly for dialog
// boxes because DlgProc returns a BOOL instead of an LRESULT (like WndProcs).
// The following _HANDLE_DLGMSG macro corrects the problem:
#define _HANDLE_DLGMSG(hwnd, message, fn)                    \
   case (message): return (SetDlgMsgResult(hwnd, uMsg,       \
      HANDLE_##message((hwnd), (wParam), (lParam), (fn))))
```

Appendix D - Directory And Filename Conventions

For compatibility with SourceSafe, project directories and files should be organized with the following general directory structure in mind.

```
\dev                          root level development directory
   _bin                          shared utilities
   _doc                          shared doc files
   _help                         shared help files
   _include                      shared include files
   _lib                          shared lib files

   medialab                      a project
      bin                           product and utility executables
      disk                          release disk images
      doc                           public product documentation files

      examples                      public example programs
         _media                        shared media for sample programs
         example1                      example program 1 source code
         example2                      example program 2 source code
         example'n'                    example program 'n' source code, etc.

      help                          public product help files
      include                       public product include files
      install                       install build procedures
      lib                           public product library files
      notes                         internal development notes
      incidents                     incident and bug reports

      src                           project source code,
         debug                         debug files
         release                       release files

      test                          test programs source code
         _media                        shared media for test programs
         test1                         test program 1 source code
         test2                         test program 2 source code
         test'n'                       test program 'n' source code, etc.

      utilities                     utility programs source code
         util1                         utility program 1 source code
         util2                         utility program 2 source code
         util'n'                       utility program 'n', etc.
```

Valid Folder and File Name Characters

For compatibility with Windows and other systems, file and folder names should be less than 31 characters (including extensions), should not start with a digit or period, and should only contain the following characters:

a-z **A-Z** **0-9** **.** (period) **_** (underscore) **-** (dash)

Avoid Spaces in Folder and File Names

Referencing hyperlinks to file names with spaces can be problematic. For this reason, using spaces within file names is not recommended. Leave out the spaces, or use underbars "_" or dashes "-" instead.

Use Lowercase Folder and File Names

In referencing a file, casing may not be obvious and, depending on the operating system, may or may not be unique. Folder names and file names should be all lower case, using underscores or dashes if necessary.

Appendix E - C Source File Template

```
/**************************************************************************
 *  template.c  -  Metagraphics C Source File Template                    *
 *                                                                        *
 *     Copyright (c) 2000 Metagraphics Corporation - All Rights Reserved  *
 *                                                                        *
 *  The source code contained herein includes proprietary information of  *
 *  Metagraphics Corporation.  Use of this source code is strictly limited *
 *  under the terms of the Metagraphics Software License Agreement.  This *
 *  source code may not be reproduced, copied or distributed, in whole or *
 *  in part, without the prior written consent of Metagraphics Corporation. *
 *   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  *
 *  Description: A brief description can go here if needed                *
 *  See Also:    MetagraphicsCodingGuide.pdf                              *
 **************************************************************************
 */

/* #include Libraries */

/* #include Interfaces */

/* STATIC VARIABLES (variables local to this file only)                */

/* EXTERNAL VARIABLES (defined elsewhere and global to all files)      */

/* STATIC FUNCTION PROTOTYPES (functions local to this file only)      */

/* GLOBAL FUNCTION IMPLEMENTATIONS                                      */

/* STATIC FUNCTION IMPLEMENTATIONS (functions local to this file only) */

/*-------------------------------------------------------------------------
 *  FunctionName() - one line description
 *
 *  Description:
 *      Multiple line description of function, parameters and return value.
 *
 *  Comments:
 *      Additional comments or implementation notes.
 *-------------------------------------------------------------------------
 */
extern                        /* ('extern' or 'static' specifier) */
MRESULT FunctionName(         /* return, result code (0=success)  */
        {type}   parameter1;  /* input,  parameter1 description   */
        {type}   parameter2;  /* in/out, parameter2 description   */
        {type}   parameter3 ) /* output, parameter3 description   */
{
    CodeGoesHere();

} /* FunctionName() */


/* End of File - template.c */
```

Appendix F - C Header File Template

```
/****************************************************************************
 *  template.h  -  Metagraphics C Header File Template                       *
 *                                                                          *
 *     Copyright (c) 2000 Metagraphics Corporation - All Rights Reserved     *
 *                                                                          *
 *  The source code contained herein includes proprietary information of     *
 *  Metagraphics Corporation.  Use of this source code is strictly limited   *
 *  under the terms of the Metagraphics Software License Agreement.  This    *
 *  source code may not be reproduced, copied or distributed, in whole or    *
 *  in part, without the prior written consent of Metagraphics Corporation.  *
 *   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  *
 *  Description: A brief description can go here if needed                  *
 *  See Also:    MetagraphicsCodingGuide.pdf                                 *
 ****************************************************************************
 */

#ifndef  TEMPLATE_H  /*=====================================================*/
#define  TEMPLATE_H  /* (don't include twice) */

#pragma once

/*#include's for other header files */

#ifdef   __cplusplus
extern "C" {            /* - - - - - - - - - - - - - - - - - - - - - - - - - */
#endif /*__cplusplus*/

/* constants */

/* macros */

/* typedefs */

/* global variables */

/* function prototypes */

#ifdef   __cplusplus
}                       /* - - - - - - - - - - - - - - - - - - - - - - - - - */
#endif /*__cplusplus*/

#endif /*TEMPLATE_H  /*=====================================================*/

/* End of File - template.h */
```

Appendix G - C++ Source File Template

```
//***************************************************************************
//  template.cpp  -  Metagraphics C++ Source File Template                  *
//                                                                          *
//     Copyright (c) 2000 Metagraphics Corporation - All Rights Reserved    *
//                                                                          *
//  The source code contained herein includes proprietary information of    *
//  Metagraphics Corporation.  Use of this source code is strictly limited  *
//  under the terms of the Metagraphics Software License Agreement.  This   *
//  source code may not be reproduced, copied or distributed, in whole or   *
//  in part, without the prior written consent of Metagraphics Corporation.  *
//   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  *
//  Description: A brief description can go here if needed                  *
//  See Also:    MetagraphicsCodingGuide.pdf                                *
//***************************************************************************

// #include Libraries

// #include Interfaces

// static variables

// static function prototypes

// public methods

// private methods

// static functions

/*--------------------------------------------------------------------------
 *  FunctionName() - one line description
 *
 *  Description:
 *      Multiple line description of function, parameters and return value.
 *
 *  Comments:
 *      Additional comments or implementation notes.
 *--------------------------------------------------------------------------
 */
extern                             // ('extern' or 'static' specifier)
MRESULT ClassName :: MethodName(   // return, result code (0=success)
        {type}    parameter1;      // input,  parameter1 description
        {type}    parameter2;      // in/out, parameter2 description
        {type}    parameter3 )     // output, parameter3 description
{
    CodeGoesHere();

} // MethodName()


/* End of File - template.cpp */
```

Appendix H - C++ Header File Template

```
//*****************************************************************************
//  template.hpp  -  Metagraphics C++ Header File Template                    *
//                                                                            *
//     Copyright (c) 2000 Metagraphics Corporation - All Rights Reserved      *
//                                                                            *
//  The source code contained herein includes proprietary information of      *
//  Metagraphics Corporation.  Use of this source code is strictly limited    *
//  under the terms of the Metagraphics Software License Agreement.  This     *
//  source code may not be reproduced, copied or distributed, in whole or     *
//  in part, without the prior written consent of Metagraphics Corporation.   *
//   - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -  *
//  Description: A brief description can go here if needed                    *
//  See Also:    MetagraphicsCodingGuide.pdf                                  *
//*****************************************************************************

#ifndef  TEMPLATE_HPP  //=====================================================
#define  TEMPLATE_HPP  // (don't include twice)

#pragma once

// include libraries

// constants

// macros

// typedefs

// class declarations:

// one line class description
class ClassName : public COtherClass
{
   // public methods

   // public variables

   // protected variables

   // protected methods

   // private methods

   // private variables

}; // class ClassName


#endif //TEMPLATE_HPP  //=====================================================

/* End of File - template.hpp */
```

Appendix I - Writing Code for ASCII & Unicode Language Portability

Writing for ASCII & Unicode Language Portability

As the need to broaden applications onto new platforms and into new markets expands, designing code for ASCII and Unicode language portability becomes a growing importance. Just as familiarity in using `int`, `short` and `long` integer types is important for designing platform portable code, basic familiarity in handling different character types is important for designing language portable code.

While many compiler and operating systems remain 8-bit ASCII orientated, a growing number of platforms are now also using 16-bit Unicode as a language standard. To be truly platform independent your C/C++ application needs to be capable of compiling and running in both ASCII and Unicode based environments. In addition, there will be cases when working on an ASCII-based platform where you may need to handle Unicode-specific text, and vice-versa on a Unicode-based platform where you may need to handle ASCII-specific text. The desired goal is to maintain a single source code base that is portable to any platform, and that supports both specific ASCII and Unicode needs when required.

Similar to size-specific `INT16(short)`, `INT32(long)` and generic `INT(int)` types for integer uses, the basis for language portability for text starts with the definition of three basic character types: size-specific `CHAR` (8-bit), `WCHAR` ("wide" char 16-bit, `wchar_t`), and generic `TCHAR` (conditional 8- or 16-bit).

Data Type	Win32 Type	Description
CHAR	`char`	8-bit ASCII character
WCHAR	`wchar_t`	16-bit Unicode character
⇔**TCHAR**	`char` or `wchar_t`	8- or 16-bit character, depending if "`UNICODE`" is defined

⇔ Indicates variable size platform-dependent conditional data type.

"**`_UNICODE`**" or "**`UNICODE`**" ? In working with many systems, you will see conditional `#ifdef`'s and `#ifndef`'s based on identifiers "`_UNICODE`" and "`UNICODE`" (either with and without a leading underbar). Normally symbols starting with an underbar are reserved for ANSI C identifiers, but in practice both "`_UNICODE`" and "`UNICODE`" are largely used interchangeably for Unicode conditionals. To keep these identifiers in sync, header files referencing these keywords usually start with the following conditional defines (contained in `mgstring.h`):

```
/* synchronize UNICODE identifiers */
#if defined(_UNICODE) && !defined(UNICODE)
#define  UNICODE
#endif

#if defined(UNICODE) && !defined(_UNICODE)
#define _UNICODE
#endif
```

In many cases we've also chosen to doubly define and drop leading underbars from other identifiers such as `_TCHAR` and `_TEXT`. The more important issue is consistency, readability and simplicity. Trying to remember which identifiers start with an underbar and which don't is one less headache to avoid.

CHAR, WCHAR and TCHAR types

CHAR is the 8-bit ASCII-specific character type, and WCHAR ("wide char") is a 16-bit Unicode-specific character type. TCHAR (generic "text char") is a platform dependent character type that is conditionally equal to either CHAR on ASCII platforms, and equal to WCHAR on Unicode platforms. The defined identifier "UNICODE" is used to identify if the native environment is a Unicode based platform. If UNICODE is undefined, then TCHAR is defined equated to ASCII CHAR; if _UNICODE is defined, TCHAR is equated to Unicode WCHAR.

```
typedef  char            CHAR;   /* 8-bit ASCII character                 */
typedef  unsigned short  WCHAR;  /* 16-bit Unicode character (wchar_t)    */

#ifndef  UNICODE
typedef  CHAR            TCHAR;  /* platform is 8-bit ASCII characters    */
#else /*ifdef UNICODE*/
typedef  WCHAR           TCHAR   /* platform is 16-bit Unicode characters */
#endif /*UNICODE*/
```

Literal Characters

The standard C/C++ single-quote (') method for specifying a single literal character works for all three character data types:

```
CHAR   charASCII   = 'A';   /* this is an 8-bit ASCII character       */
WCHAR  charUnicode = 'B';   /* this is a 16-bit Unicode character     */
TCHAR  charSystem  = 'C';   /* ASCII or Unicode depending on platform */
```

The variable charUnicode will be a 16-bit value 0x0042, which is the Unicode representation for the letter B. (Keep in mind that Intel processors store multibyte values with the least significant bytes first, so the bytes are actually stored in memory in the sequence 0x42, 0x00 - remember this when examining a hex dump of Unicode text in memory.)

Literal Strings

Literal ASCII CHAR Strings

Using the standard C/C++ double-quote (") method for specifying literal strings works for ASCII only, but will not work for Unicode character strings.

```
CHAR  strASCII[] = "this is an ASCII string of 8-bit characters";
```

Literal Unicode WCHAR Strings

The ANSI C extension for defining literal Unicode strings is to precede the first double-quote with the capital letter L (as in "Long"). The L preceding the first double-quote is required, and there cannot be any spaces between the L and the first double-quote. The L tells the compiler that you want the string to be stored as 16-bit WCHAR characters.

```
WCHAR strUnicode[] = L"this is a Unicode string of 16-bit characters";
```

Literal Generic TCHAR Strings

For the conditional TCHAR character type, we need a method to conditionally define strings either as an 8-bit ASCII CHAR string, or as a 16-bit Unicode WCHAR string. A method to handle this is to define a special TEXT() macro that performs this function.

```
        #ifndef  UNICODE
        #define  __T(s)  s       /* platform is ASCII   */
        #else  /* ifdef UNICODE */
        #define  __T(s)  L##s    /* platform is Unicode */
        #endif /*UNICODE*/

        #define  TEXT(s)  __T(s)
```

`L##s` uses the ANSI C `##` "token paste" operator to have the C preprocessor concatenate the letter `L` with the token quoted string `s`. With the above `#define TEXT()` macro we can now specify `TCHAR` strings that are conditionally either ASCII or Unicode based on the target platform:

```
    TCHAR strSystem[] = TEXT("ASCII or Unicode string depending on platform");
```

String Format Conversion Functions

For non-literal text, format conversion functions can be used when needed to convert 8-bit ASCII text to 16-bit Unicode, or vice-versa to convert 16-bit Unicode text to 8-bit ASCII. In converting 16-bit Unicode to 8-bit ASCII, of course, Unicode characters above 255 must be stripped to the smaller 8-bit ASCII range. For simplicity, Unicode characters above 255 are usually simply converted to ASCII "space" characters. Two functions are used for string format-conversion and copy operations:

```
   /* StrnCvtAW() - Convert 8-bit Ascii string to 16-bit Unicode string  */
   WCHAR *StrnCvtAW(       /* return, ptr to destination Unicode string, wStr */
         WCHAR  *wStr,     /* output, destination Unicode string             */
   const CHAR   *aStr,     /* input,  source Ascii string                    */
         UINT    nwChars );/* input,  # of WCHARs in destination wStr buffer  */

   /* StrnCvtWA() - Convert 16-bit Unicode string to 8-bit Ascii String  */
   CHAR  *StrnCvtWA(       /* return, ptr to destination Ascii string, aStr   */
         CHAR   *aStr,     /* output, destination Ascii string               */
   const WCHAR  *wStr,     /* input,  source Unicode string                  */
         UINT    naChars );/* input,  # of CHARs in destination aStr buffer   */
```

`StrnCvtAW()` and `StrnCvtWA()` operate similar to the standard ANSI C library string copy functions `strncpy()` and `wcsncpy()` for copying ASCII and Unicode strings, respectively. `StrnCvtAW()` and `StrnCvtWA()`, however perform the additional character format-conversion operation (implementation of `StrnCvtAW()` and `StrnCvtWA()` is provided in `mgstring.c`).

For use with the generic `TCHAR` type, simple `#define`'s are used to assign the appropriate conversion function for formatting and copying ASCII `CHAR` or Unicode `WCHAR` strings to and from generic `TCHAR` strings:

```
        #ifndef  UNICODE        /* platform/TCHAR is 8-bit ASCII   */
        #define  StrnCvtAT  strncpy     /* copy ASCII to TCHAR (ASCII)     */
        #define  StrnCvtWT  StrnCvtWA   /* copy Unicode to TCHAR (ASCII)   */
        #define  StrnCvtTA  strncpy     /* copy TCHAR (ASCII) to ASCII     */
        #define  StrnCvtTW  StrnCvtAW   /* copy TCHAR (ASCII) to Unicode   */

        #else  /* ifdef UNICODE - platform/TCHAR is 16-bit Unicode */
        #define  StrnCvtAT  StrnCvtAW   /* copy ASCII to TCHAR (Unicode)   */
        #define  StrnCvtWT  wcsncpy     /* copy Unicode to TCHAR (Unicode) */
        #define  StrnCvtTA  StrnCvtWA   /* copy TCHAR (Unicode) to ASCII   */
        #define  StrnCvtTW  wcsncpy     /* copy TCHAR (Unicode) to Unicode */
        #endif /*UNICODE*/
```

The six format conversion functions outlined above provide the means to convert and copy any string type, ASCII, Unicode or generic `TCHAR`, to any other string type.

String Library Functions

In addition to the basic character types, literal specifiers and format-conversion functions, we also need support for common ANSI C library string manipulation functions. The latest ANSI C `string.h` header file fortunately includes library definitions supporting functions for both 8-bit ASCII and 16-bit Unicode. Similar to the `strlen()` function that returns the number of characters in an ASCII string, ANSI C now also provides a `wcslen()` function that returns the number of characters in a Unicode string (this is very important since the ASCII `strlen()` function will not return the proper length of a Unicode string!). There is a similar matching Unicode function for most of the standard ASCII string functions. For use our generic `TCHAR` type, a third set of functions needs to be defined based on the "`UNICODE`" keyword. The following table summarizes the data types and function names provided through `mgstring.h` for each of the type-specific `CHAR` and `WCHAR` types, and also for our generic `TCHAR` type.

Trying to make sense of the names for the standard C string library functions can be a challenge at times. Unicode function names add further complications by embedding a "w" ("wide" designation) sometimes at the beginning, sometimes in the middle, and sometimes at the end of the ASCII function name. With over 100 ASCII and Unicode library function names to deal with, adding another 50+ names for generic `TCHAR` functions makes C string library names begin to look like alphabet soup.

For simplicity we've adopted a mixed-case naming convention based on the original ASCII function names that most C/C++ programmers already know. Mixed-casing helps make the names more readable and easier to remember. To differentiate type-specific ASCII and Unicode functions, the mixed-case names are simply appended with an "A" or "W" depending if the function takes ASCII or Unicode arguments. For generic `TCHAR` functions, where the function arguments may be either ASCII or Unicode (depending if "`UNICODE`" is defined or not), the function names are defined without a specific appended type.

In certain cases some names have also been expanded for better clarity - for example, `atoi()` and `_wtoi()` were renamed `StrToIntA()`, `StrToIntW()` and `StrToInt()` for ASCII, Unicode and generic `TCHAR` functions, respectively. While you can still use the original standard C string library function names, the following function names defined through `mgstring.h` are generally easier to read, understand and remember.

	ASCII	**Unicode**	**Generic**
character size	`8-bit`	`16-bit`	`8- or 16-bit`
type	`CHAR`	`WCHAR`	`TCHAR`
literal character	`'.'`	`'.'`	`'.'`
literal string	`"..."`	`L"..."`	`TEXT("...")`
get character string length	**`StrLenA()`** `strlen()`	**`StrLenW()`** `wcslen()`	**`StrLen()`** `StrLenT()`
find character in string	**`StrChrA()`** `strchr()`	**`StrChrW()`** `wcschr()`	**`StrChr()`** `StrChrT()`
find character, ignore case	**`StriChrA()`** `_strichr()`	**`StriChrW()`** `_wcsichr()`	**`StriChr()`** `StriChrT()`
reverse-find character	**`StrrChrA()`** `strrchr()`	**`StrrChrW()`** `wcsrchr()`	**`StrrChr()`** `StrrChrT()`
find substring	**`StrStrA()`** `strstr()`	**`StrStrW()`** `wcsstr()`	**`StrStr()`** `StrStrT()`
copy string	**`StrCpyA()`** `strcpy()`	**`StrCpyW()`** `wcscpy()`	**`StrCpy()`** `StrCpyT()`

copy string, w/max	**StrnCpyA()**
	strncpy()
		StrnCpyW()
		wcsncpy()
			StrnCpy()
			StrnCpyT()

convert string, ASCII to ...	StrnCpyA()	**StrnCvtAW()**	**StrnCvtAT()**
convert string, Unicode to ...	**StrnCvtWA()**	StrnCpyW()	**StrnCvtWT()**
convert string, TCHAR to ...	**StrnCvtTA()**	**StrnCvtTW()**	StrnCpy()

concatenate string	**StrCatA()**
	strcat()
		StrCatW()
		wcscat()
			StrCat()
			StrCatT()

concatenate string, w/max	**StrnCatA()**
	strncat()
		StrnCatW()
		wcsncat()
			StrnCat()
			StrCatT()

compare string	**StrCmpA()**
	strcmp()
		StrCmpW()
		wcscmp()
			StrCmp()
			StrCmpT()

compare string, w/max	**StrnCmpA()**
	strncmp()
		StrnCmpW()
		wcsncmp()
			StrnCmp()
			StrnCmpT()

compare string, ignore case	**StriCmpA()**
	_stricmp()
		StriCmpW()
		_wcsicmp()
			StriCmp()
			StriCmpT()

compare string, nocase, max	**StrniCmpA()**
	_strnicmp()
		StrniCmpW()
		_wcsnicmp()
			StrniCmp()
			StrniCmpT()

get count of matching chars	**StrSpnA()**
	strspn()
		StrSpnW()
		wcsspn()
			StrSpn()
			StrSpnT()

get matching char index	**StrcSpnA()**
	strcspn()
		StrcSpnW()
		wcsspn()
			StrcSpn()
			StrcSpnT()

find next token	**StrTokA()**
	strtok()
		StrTokW()
		wcstok()
			StrTok()
			StrTokT()

locate matching character	**StrpBrkA()**
	strpbrk()
		StrpBrkW()
		wcspbrk()
			StrpBrk()
			StrpBrkT()

is alphanumeric character?	**IsAlnumA()**
	isalnum()
		IsAlnumW()
		iswalnum()
			IsAlnum()
			IsAlnumT()

is alpha character?	**IsAlphaA()**
	isalpha()
		IsAlphaW()
		iswalpha()
			IsAlpha()
			IsAlphaT()

is decimal digit (0-9)?	**IsDigitA()**
	isdigit()
		IsDigitW()
		iswdigit()
			IsDigit()
			IsDigitT()

is hex digit (0-9, A-F, a-f)?	**IsHexDigitA()**
	isxdigit()
		IsHexDigitW()
		iswxdigit()
			IsHexDigit()
			IsHexDigitT()

is lowercase character?	**IsLowerA()**
	islower()
		IsLowerW()
		iswlower()
			IsLower()
			IsLowerT()

is uppercase character?	**IsUpperA()**
	isupper()
		IsUpperW()
		iswupper()
			IsUpper()
			isUpperT()

is white-space character?	**IsSpaceA()**
	isspace()
		IsSpaceW()
		iswspace()
			IsSpace()
			IsSpaceT()

is printable character?	**IsPrintA()**
	isprint()
		IsPrintW()
		iswprint()
			IsPrint()
			IsPrintT()

is punctuation character?	**IsPunctA()**
	ispunct()
		IsPunctW()
		iswpunct()
			IsPunct()
			IsPunctT()

convert char to lowercase	**ToLowerA()**
	tolower()
		ToLowerW()
		towlower()
			ToLower()
			ToLowerT()

convert char to uppercase	**ToUpperA()**
	toupper()
		ToUpperW()
		towupper()
			ToUpper()
			ToUpperT()

convert integer to string	**IntToStrA()**
	_itoa()
		IntToStrW()
		_itow()
			IntToStr()
			IntToStrT()

convert long to string	**LongToStrA()**
	_ltoa()
		LongToStrW()
		_ltow
			LongToStr()
			LongToStrT()

convert string to integer	**StrToIntA()** atoi()	**StrToIntW()** _wtoi()	**StrToInt()** StrToIntT()
convert string to long	**StrToLongA()** atol()	**StrToLongW()** _wtol()	**StrToLong()** StrToLongT()
format data to stdout	**PrintfA()** printf()	**PrintfW()** wprintf()	**Printf()** PrintfT()
format data to file	**FPrintfA()** fprintf()	**FPrintfW()** fwprintf()	**FPrintf()** FPrintfT()
format data to string	**SPrintfA()** sprintf()	**SPrintfW()** swprintf()	**SPrintf()** SPrintfT()
format data to string, w/max	**SNPrintfA()** _snprintf()	**SNPrintfW()** _snwprintf()	**SNPrintf()** SNPrintfT()
format string, arglist-ptr	**VPrintfA()** vprintf()	**VPrintfW()** vwprintf()	**VPrintf()** VprintfT()
format arglist-ptr to file	**VFPrintfA()** vfprintf()	**VFPrintfW()** vfwprintf()	**VFPrintf()** VFPrintfT()
format arglist-ptr to string	**VSPrintfA()** vsprintf()	**VSPrintfW()** vswprintf()	**VSPrintf()** VSPrintfT()
format arg-ptr to str, w/max	**VSNPrintfA()** _vsnprintf()	**VSNPrintfW()** _vsnwprintf()	**VSNPrintf()** VSNPrintfT()
get current working directory	**GetCwdA()** _getcwd()	**GetCwdW()** _wgetcwd()	**GetCwd()** GetCwdT()
open file	**FOpenA()** fopen()	**FOpenW()** wfopen()	**FOpen()** FopenT()
read formatted from stdin	**ScanfA()** scanf()	**ScanfW()** wscanf()	**Scanf()** ScanfT()
read formatted from string	**SScanfA()** sscanf()	**SScanfW()** swscanf()	**SScanf()** SscanfT()
read formatted from file	**FScanfA()** fscanf()	**FScanfW()** fwscanf()	**FScanf()** FscanfT()
read character from stdin	**GetCharA()** getchar()	**GetCharW()** getwchar()	**GetChar()** GetCharT()
read character from file	**GetcA()** getc()	**GetcW()** getwc()	**Getc()** GetcT()
read line from stdin	**GetsA()** gets()	**GetsW()** _getws()	**Gets()** GetsT()
read line from file	**FGetsA()** fgets()	**FGetsW()** fgetws()	**FGets()** FGetsT()
write character to stdout	**PutCharA()** putchar()	**PutCharW()** putwchar()	**PutChar()** PutCharT()
write character to file	**PutcA()** putc()	**PutcW()** putwc()	**Putc()** PutcT()
write string to stdout	**PutsA()** puts()	**PutsW()** _putws()	**Puts()** FGetsT()
write string to file	**FPutsA()** fputs()	**FPutsW()** fputws()	**FPuts()** FPutsT()

With our `CHAR`, `WCHAR` and `TCHAR` types, along with our character and string literal specifiers, and the string library functions outlined above, we have a cohesive and portable system for handling characters and text on any ASCII or Unicode platform. Also where needed, we have our type-specific functions both for handling ASCII-specific strings on Unicode platforms, and vice-versa for handling Unicode-specific strings on ASCII platforms.

MetaWINDOW ASCII/Unicode Functions

Metagraphics MetaWINDOW provides both type-specific and generic functions for `CHAR`, `WCHAR` and `TCHAR` data types. (For 16-bit `WCHAR` and `TCHAR` Unicode types, the associated `...W` and `...T` functions automatically perform Unicode to glyph position translation.)

	ASCII	**Unicode**	**Generic**
type	`CHAR`	`WCHAR`	`TCHAR`
get character width	`CharWidth()`	`CharWidthW()`	`CharWidthT()`
get string width	`StringWidth()`	`StringWidthW()`	`StringWidthT()`
draw character	`DrawChar()`	`DrawCharW()`	`DrawCharT()`
draw string	`DrawString()`	`DrawStringW()`	`DrawStringT()`

Support for earlier glyph-position strings is also provided:

	ASCII (8-bit)	**Glyph (16-bit)**	**Generic (8/16)**
type	`CHAR`	`USHORT`	`TCHAR`
get character width	`CharWidth()`	`CharWidth16()`	
get string width	`StringWidth()`	`StringWidth16()`	
draw character	`DrawChar()`	`DrawChar16()`	
draw string	`DrawString()`	`DrawString16()`	

TypeServer ASCII/Unicode Functions

Metagraphics TypeServer provides both type-specific and generic functions for `CHAR`, `WCHAR` and `TCHAR` data types. (TypeServer C function names are prefixed with "`tsStrike_`"; C++ method names are unique within the `tsCStrike` class.)

	ASCII	**Unicode**	**Generic**
type	`CHAR`	`WCHAR`	`TCHAR`
get char dimensions	`GetCharExtentA()`	`GetCharExtentW()`	`GetCharExtentT()`
get string dimensions	`GetStringExtentA()`	`GetStringExtentW()`	`GetStringExtentT()`
draw character	`DrawCharA()`	`DrawCharW()`	`DrawCharT()`
draw string	`DrawStringA()`	`DrawStringW()`	`DrawStringT()`

mgstring.h/mgstring.c

Copies of `mgstring.h` and `mgstring.c` may be downloaded from the Metagraphics web site at:

http://www.metagraphics.com/pubs/mgstring.zip

Appendix J - Resources

Programming Conventions

Books

- *"Enough Rope To Shoot Yourself In The Foot – Rules for C and C++ Programming"*, Allen Holub, McGraw-Hill, ISBN: 0-07-029689-8
- *"Code Complete"*, Steve McConnel, Microsoft Press, ISBN: 1-55615-484-4
- *"Effective C++"*, Scott Meyers, Addison-Wesley, ISBN: 0-201-56364-9
- *"More Effective C++"*, Scott Meyers, Addison-Wesley, ISBN: 0-201-63371-X
- *"C++ Strategies and Tactics"*, Robert Murray, Addison-Wesley, ISBN: 0-201-56382-7
- *"Extreme Programming Installed"*, Ron Jeffries, Addison-Wesley, ISBN: 0-201-70842-6
- *"Extreme Programming Explained"*, Kent Beck, Addison-Wesley, ISBN: 0-201-61641-6

Internet

- Taligent Guide to Developing Programs, http://hpsalo.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/books/WM/WM_1.html
- Capability Maturity Model for Software, http://www.sei.cmu.edu/cmm/cmm.html
- Extreme Programming, http://www.extremeprogramming.org/
- Mozilla C++ Portability Guide, http://mozilla.org/hacking/portable-cpp.htm

Code Optimization

Profilers

- Microsoft *Visual C++ Profiler*
- Intel *VTune*
- NuMega *TrueTime*

Books

- *"Inner Loops"*, Rick Booth, Addison-Wesley, ISBN: 0-201-47960-5
- *"DirectX, RDX, RSX and MMX Technology*", Coelho & Hawash,, ISBN: 0-201-30944-1
- *"Zen of Code Optimization"*, Michael Abrash, Coriolis Books, ISBN: 1-883577-03-9

Internet

- rec.games.programmer
- comp.graphic.api.opengl
- OpenGL and DirectX game programming List

Windows Programming

Books

- *"Programming Applications for Microsoft Windows 2000, Fourth Edition",* Jeffrey Richter, Microsoft Press, ISBN: 1-57231-996-8
- *"Programming Windows, Fifth Edition"*, Charles Petzold, Microsoft Press, ISBN: 1-57231-995-X
- *"Advanced Windows",* Jeffrey Richter, Microsoft Press, ISBN: 1-57231-548-2
- *"Win32 Programming"*, Brent Rector, Addison-Wesley, ISBN: 0-201-63492-9
- *"Windows 2000 Graphics API Black Book"*, Damon Chandler, Coriolis Books, ISBN: 1-57610-876-7
- *"Windows 98 API Programming for Dummies"*, Namir Shammas, IDG Books, ISBN: 0-7645-030-0
- *"Windows 98 – A Developer's Guide"*, Jeffrey Richter, M&T Books, ISBN: 1-55851-418-X
- *"Windows Internals",* Matt Pietrek, , ISBN: 0-2001-62217-3