Adapting MSP to Microsoft C++

1. **Introduction**
   a. **General comments**
      i. **Purpose**
         (1) MSP was developed in a PC environment with the original version of Borland C++, then adapted in several stages to Version 5.0, the standard for this book. Many readers, however, prefer to or must use a different compiler. The environment most frequently mentioned that is somewhat close to the book's is Microsoft's. This file describes my adaptation of MSP as a suite of Microsoft Visual C++ Win32 Console Applications.
         (2) It's an adaptation guide, not a completed product. Most MSP features described in this book work under the adaptation, but it has not been used for further development. With it, you can see how these features work in your Microsoft environment. To use it for a major project, however, you'll probably want to select only the appropriate MSP modules. You'll need to test the adaptation much more thoroughly, and probably add minor corrections as needed by your specific application.
      ii. **Major differences**
         (1) Microsoft's complex mathematics module is implemented with templates. That adds more levels of implicit conversions for the compiler to consider when resolving overloaded function declarations. Many more ambiguities would result, and some methods MSP uses to resolve them would no longer work. This would cause so many changes that you'd lose sight of the rest. So the adaptation contains a completely rewritten complex module.
         (2) Microsoft handles hardware faults, such as overflow detected by the floating-point processor, by throwing"structured exceptions". It provides no way to intervene in this process, and forbids simultaneous use of structured exceptions and the C++ standard exceptions that MSP uses. Converting all MSP exception handling would cause too many changes, and would be a step backward in programming style. Therefore the adapation does not handle hardware exceptions.
      iii. **Minor differences**
         (1) Differences in the stages of C++ evolution on which the compilers are based cause minor syntax adaptations, some of which occur frequently.
         (2) Microsoft's rules for generating object code for needed instances of template functions differ from Borland's. The source code codas must be expanded to force generation.
         (3) Microsoft's DOS execution windows differ slightly from Borland's, and require some adjustment of output.
         (4) There are various minor differences beyond these.

iv. **Adaptation status**

(1) The adaptation was designed to implement as many of this book's MSP features as possible. It goes no farther than the demonstration and test programs included on the CD-ROM packaged with the book. It needs more testing to reveal bugs these client programs won't detect.

(2) Each MSP module's source code file has a coda consisting of otherwise useless functions that invoke MSP functions as required, ensuring that appropriate MSP function instances are generated in the object code. Microsoft's requirements differ from Borland's: codas in the adaptation are more extensive than in the Borland version. I couldn't find Microsoft's rules, so the adapted codas invoke only instances required by the CD-ROM's demonstration and test programs. Your own programs will almost certainly invoke further instances, and linker errors will result. Find the appropriate MSP source code files, and add to their codas the required invocations.

(3) Hardware exceptions—particularly those detected by the floating-point processor—are not handled by the adaptation. Microsoft's method for providing this capability is incompatible with this book's exception handling mechanism. It was judged more important to keep the rest of the book's features than to change all of them to Microsoft's less graceful mechanism.

(4) The Microsoft Win32 non-console Application environment (like Borland's C++ Builder) is incompatible with this adaptation. Its exception handling mechanism is still farther from the book's. Moreover, it uses Standard C++ headers, which include many subtle differences I haven't yet researched.

v. **Overall assessment** This adaptation was painful. The Microsoft user interface is continually annoying, much less convenient than Borland's. Documentation is less adequate. The lack of a graceful way to intervene in hardware exception handling is especially serious. Microsoft's policies for generating object code for function instances, and for resolving ambiguous function invocations, are inadequate and insufficiently documented. These problems are interrelated: you often can't tell whether you violated a C++ language rule, Microsoft didn't implement it properly, you didn't include the correct invocation in a coda, Microsoft didn't generate code when it should have, or . . . And the user interface doesn't help answer these questions. This book would probably not have been written at all had I been trying to use Microsoft C++ at the onset.

b. `MicrosoftAdaptation` folder organization

i. This `AdaptingToMicrosoft` file first outlines the purpose of the adaptation, its major features, and its current status. The file then describes in detail all the changes made to the Borland C++ Version 5.0 code.

ii. The adaptation itself is contained in several subfolders organized exactly like the folders in the root directory of this diskette.

iii. One file, however, cannot be included. The adaptation requires replacing Microsoft's `Complex.H` header file and source code. I simply used Borland's, stripped of features peculiar to that environment, and only slightly modified beyond that. Later in this file you'll find instructions for that phase of the adaptation process. It was in fact the easiest part for me to carry out. But distributing the adapted file would violate copyright restrictions and ethics. Borland's design is elegant and effective. If you must use Microsoft's environment, buy Borland's product and adapt this feature as I did. If you encounter severe difficulty, email me at `smith@math.sfsu.edu`.

c. `AdaptingToMicrosoft` file
The next heading in this file describes a few **systematic changes** involved in converting from the Borland to the Microsoft evironment. Subsequent headings are included for each MSP module. Under each heading you'll find the list of source code changes for that module and for the related demonstration and test programs, except for the few systematic changes already detailed.

2. **Systematic changes** (These won't be mentioned again.)
   a. Set the tab size and indentation to 2 instead of Microsoft's default 4.
   b. Change numerous occurrences of the unprintable characters ² and ± to `^2` and `+or-`, and maintain vertical output alignment.
   c. Remove `#pragma warn` lines. They have no effect.
   d. Remove outer parentheses from all instances of `throw(Exception( ));` This seems to be a Microsoft compiler bug; it severely affects compiling efficiency.
   e. Append newlines and `Pause()` after the last outputs of many test and demonstration programs, so that Microsoft's exit message doesn't interfere with program output, and the console window stays open for you to see program output.
   f. Remove `#include <Excpt.H>` statements.

3. **General module**
   a. `Fig2_03` No further changes.
   b. `Fig2_04` No further changes.
   c. `Fig3_06`
      i. This program doesn't demonstrate what was intended. It outputs `32769 32769` because the compiler uses 32-bit integers. To demonstrate what was intended, put $2^{31} + 1$ in place of $32769 = 2^{15} + 1$.
   d. `General.H` and `General.CPP`
      i. The book includes function `min(int,int)` in this module. Add the analogous function `max(int,int)`. Probably, the analogous `double` functions are required, too, but during adaptation, the macro versions mentioned condescendingly in the text were substituted for invocations of the double versions instead.

ii. Add `void randomize()` to `General.H` and define this in `General.CPP` as `srand((unsigned)time(NULL));` following a Microsoft example. Add `#include <Time.H>` to `General.CPP`.

e. `GenTest`

i. <Alt-F4> doesn't break out of the test loop and halt. You have to kill the program with the exit button. Change the cue.

4. **Scalar module**

a. `Scalar.H` and `Scalar.CPP`

i. Change `exception` to `_exception` in `_matherr`. Add `#include <Math.H>` for the definition of `_exception`.

ii. Append `return 1` at end of `_matherr`.

iii. In `ForShow` change

```
try  { S = new char[80]; }
catch(xalloc) {
```

to

```
S = new char[80];
if (S == NULL) {.
```

Remove `#include <Except.H>`: it's no longer needed.

iv. Microsoft evidently doesn't permit defaulted arguments in template functions. Remove the defaults from the second arguments of `Show` and `ForShow`, and add versions of these functions with just the first argument to both `Scalar.H` and `Scalar.CPP`. Adjust the coda of the latter accordingly.

v. As reported later, complex number features are changed. In `Scalar.H` change `<Complex.H>` to "`MSPComplex.H`" and remove `#define ii`. The latter will go in the new `MSPComplex.H`.

vi. As reported later, all references to floating point exception handling are removed from `Scalar.CPP`.

b. `ScaTest`

i. Change `<Complex.H>` to "`MSPComplex.H`".

ii. Omit "`String.H`". It's not necessary.

c. `Fig3_10` Change `<Complex.H>` to "`MSPComplex.H`".

d. `Fig3_11`

i. Change `pow10(n)` to `pow(10,n)` because the former is not defined in Microsoft's `Math.h`.

e. `Fig3_12`

i. Change `exception` to `_exception`.

ii. Microsoft's run-time error handler doesn't display any error message. The invalid `sqrt(-1)` invocation merely returns `-1.#IND`. But if you output `Hello` in the substituted `_matherr` you'll see that it's in fact executed.

f. `Fig3_14`

i. To actually run this, replace the two `#include` lines by `#include "Scalar.H"`.
ii. Microsoft C++ evidently doesn't detect `TLOSS` situations.

g. `Fig3_15`
i. Change "`Scalar.H`" to "`General.H`".
ii. Change `cos` to `Cos` except in the `return` statement.
iii. Because Microsoft C++ doesn't detect `TLOSS` situations, change `(1e70,1)` to `(1,1e70)`. Then you get an `OVERFLOW` error in `sinh`.

h. `Fig3_16`
i. The interplay between `_control87` and `_fpreset` is not clear, and neither are the Microsoft masking defaults. It's only safe to assume that any floating point error must be caught somehow if the program is to continue, because the error handler must execute `_fpreset`. I rewrote the program. It outputs `0, 0, 1.#INF, 1.#INF, -1.#IND`, and crashes on the last trial.

i. `Fig3_19`
i. Omit `#include <Signal.H>`.
ii. With no MSP hardware exception handling in `Scalar.CPP`,
(1) `a = 0` outputs `1.#INF`.
(2) `a = 1e-200` does the same.
(3) `a = 1e200` outputs `0`.
iii. MSP hardware exception handling cannot be implemented in Microsoft C++ without considerable redesign.
iv. The Microsoft run-time environment employs the obsolete *structured exception* technique. That uses `__try` and `__except` blocks with `__except` arguments in place of `try` and `catch` blocks. An unmasked hardware exception in a `__try` block transfers control to function `_fpieee_flt`, executed as the first `__except` argument. That function, declared in header `<FPIEEE.H>`, knows how to get information about the exception and transmit it to the handler function, whose address is in the second `__except` argument. This handler function then executes and does whatever you want done to all exceptions. Then control passes to the `__except` block, which has access to variables in the arguments of `_fpieee_flt`, hence to information about the exception. At this point the stack is as though it had successfully executed the erroneous operation and got a return value (specified by the handler function), but had not executed the `__try` block after the error.
v. There seems to be no alternative to this technique, so structured exceptions are required for handling hardware exceptions. But Microsoft C++ forbids mixing exception handling techniques. Therefore the entire MSP exception handling apparatus would have to be redesigned to use structured exceptions.
vi. This would require

(1) a new catalog of structured exception types, consistent with Microsoft,
(2) minor revision of the MSP `_matherr` function to raise structured exceptions appropriately,
(3) elimination of the MSP `FPEHandler` function and associated material,
(4) replacement of all `try` keywords by `__try`,
(5) design of a standard `__except` statement (necessarily rather complicated) which then will replace all `catch` statements,
(6) replacement of all `throw` statements by corresponding `RaiseException` statements.

vii. This appears to be a step backward in programming practice. The exception handling technique of Borland C++ Version 5 may not survive much longer, but probably won't be replaced by something more primitive, such as structured exceptions. Therefore, MSP hardware exception handling techniques will not be implemented in Microsoft C++. The adaptation will handle exceptions detected by the `Math.H` library functions, but not those detected only by the numeric processor.

viii. Remove the code for `FPExceptionName`, `FPEHandler`, `MSPErrorHandling`, and remove `#pragma startup MSPErrorHandling`.

j. `Fig3_25`
   i. This program's output differs in format and roundoff from the Borland version.

k. `ErrTest`
   i. The first test is irrelevant and crashes. Remove that and `#include <Signal.H>`.
   ii. Otherwise, it's OK, and `pow` does report the arguments correctly.

l. `Riemann` No further changes.

5. **Complex module**

a. Microsoft implemented the `complex` type with templates. The concomitant type conversions caused severe ambiguity problems with MSP. I decided to replace their complex mathematics module and associated object code with my own. I stripped the Borland C++ Version 5 `Complex.H` header file of features not pertinent to *C++ toolkit*, cleaned up the code, and compiled it for use here. The resulting module consists of `MSPComplex.H` and `MSPComplex.CPP`. In the MSP organization, it should be prior to `General`, parallel to C++ `double` and `int` features.

b. Move `#define ii complex(0,1)` from `Scalar.H` to `MSPComplex.H`.

c. Copy `#define M_PI` and `#define M_LOG10E` from Borland's `Math.H` to `MSPComplex.H`. Microsoft evidently doesn't provide these constants.

d. The default constructor compiled but wouldn't link. Moreover, the compiler didn't supply one. The constructor with real and imaginary arguments, both defaulted to zero, suffices.

e. Some + and - operators compiled but didn't link. Evidently Microsoft doesn't compile object code unless it's used. So MSPComplex.CPP has a brief coda using some of its functions.

6. **ElemFunc module**
   a. ElemFunc.H and ElemFunc.CPP
      i. Change Values.H to Limits.H and MAXLONG to LONG_MAX.
   b. ElemTest No further change.

7. **Equate1 module**
   a. Equate1.H and Equate1.CPP
      i. In both files, replace the template NR by its double and complex instances.
   b. Equ1Test
      i. Split templates FdFforCam, Bouncer, and NRTest into double and complex instances, because Microsoft can't settle some ambiguity of the template parameters.

8. **Vector module**
   a. VectorL.H and VectorL.CPP
      i. Replace the unprintable empty vector symbol *N* in ForShow by the string "EMPTY".
      ii. Replace the three overloaded = declarations in Vector.H by one, that does not permit any type conversion. Modify the code in VectorL.CPP accordingly. Microsoft doesn't recognize the variants, and would incorrectly substitute the default assignment operator.
      iii. Supply a new nonmember overloaded function void Assign(Vector<Scalar>&,const Vector<double>&), which acts like the converting assignment operator function.
      iv. In SetUp change from the xalloc method of handling a new exception. Use the fact that new returns NULL in case of exception. Eliminate the outer try block.
      v. In the catchall block in SetUp change High and Low to Hi and Lo. This is a bug in original MSP code.
      vi. In KeyIn and MakeUnit add return This at end. (They're never executed.)
      vii. The console screen is 80 characters wide, but you get an automatic newline if there's anything in column 80. Change T.screenwidth in Show to ScreenWidth, and remove its calculation.
      viii. Microsoft wouldn't generate some instances of Vector template functions until I added a list of them to UseVectorL.
   b. VecTestL

i. Microsoft wouldn't generate some instances of `Vector` template functions until I added kludge function `Use_cin_to_Vector`.
ii. Test function `Assign` in place of the mixed type assignment operator.
iii. <Ctrl-z> to end `Test2` doesn't always work right.
iv. `Test4` reveals that when attempting to open the file just renamed, Microsoft doesn't flush a buffer. It doesn't realize that the file is gone. It seems then to create an empty file with the old name. It attempts to read from the empty file, and throws and catches an exception. That leads to a `return`, which ends the test. The adaptation will not attempt to fix this situation. It requires greater understanding of Microsoft file handling.
v. `Test5` stopped showing `15874` on my machine. Perhaps the last output statement of the `catch` block didn't execute.

c. `VectorM.H` and `VectorM.CPP`
i. In `MaxNorm` in `VectorM.CPP` change `max` to `__max`.

d. `VecTestM` No further changes.
e. `Stokes` No further changes.

9. **Polynom module**

a. `Polynom.H` and `Polynom.CPP`
i. Remove the `double` to `complex` conversion overload of the assignment operator, and the `double` to `complex` constant polynomial constructor. As in the Vector module, provide instances of function `Assign` to get around the absence of the type-converting assignment operator. These are invoked in `Roots` and `Divide`.
ii. Split templates `GCD` and `LCM` into separate functions with one parameter, and with two, imitating the adaptation of `Show` and `ForShow` in the Vector module.
iii. In the `*` operator and in `LCM`, change `Zero(0)` to `Zero`.
iv. In `CauchyBound` change `max` and `min` to `__max` and `__min`.
v. The compiler didn't generate many instances of the templates for addition, subtraction, and multiplication of polynomials by scalars. These were added one by one to both `Polynom.H` and `Polynom.CPP`, and mentioned in the coda. (I know I didn't find all that should be there: I merely made `PolyTest.CPP` link. Moreover, in my experience, some later corrections usually render some earlier ones unnecessary. But I didn't go through these to check that out.)

b. `PolyTest`
i. Use `Assign` in `MultiplicitiesTest` to get around the absence of the type-converting assignment operator.
ii. I didn't wait long enough for `Test9` in `PolyTest.CPP` to run out of memory, so that exception hasn't been tested.

c. `Legendre` No further changes.

10. **Matrix module**
    a. MatrixL.H and MatrixL.CPP
        i. Replace the three overloaded = declarations in Matrix.H by one, that does not permit any type conversion. Modify the code in MatrixL.CPP accordingly. Microsoft doesn't recognize the variants, and would incorrectly substitute the default assignment operator.
        ii. Supply a new nonmember overloaded function void Assign(Matrix<Scalar>&, const Matrix<double>&), which acts like the converting assignment operator function. Mention it in the MatrixL.CPP coda.
        iii. Reverse some logic in operator[], Vector<Scalar> and Col to put return at end of each function.
        iv. Add a nonexecuted return This at the end of KeyIn.
        v. In SetUp change from the xalloc method of handling a new exception. Use the fact that new returns NULL in case of exception. Eliminate the outer try block.
        vi. Add the Use_cin_to_Vector kludge mentioned earlier under VecTestL. I don't understand this problem.
    b. MatTestL
        i. Test the Assign function in place of the mixed type assignment operator.
        ii. Replace the int i declarations within two for loops in MatTestL.CPP, by a single one outside.
        iii. Test1 and Test2 ran. Test3 didn't work right. It threw the exception when the disk got full, but apparently didn't have enough memory left to catch it. I didn't wait long enough, perhaps, the second time I filled the disk. It churned a long time while almost full. Perhaps Windows became obsessed with reorganizing its swap file. (I won't troubleshoot this, since it worked with Borland.)
    c. MatrixM.H and MatrixM.CPP
        i. Add a bunch of invocations to the coda. And invoke UseMatrixM(int). (Bug in original code.)
        ii. In RowNorm change max to __max.
    d. MatTestM No further changes.

11. **GaussEl module**
    a. GaussEl.H and GaussEl.CPP
        i. Invert some logic in Solve so that return comes last.
    b. GausTest
        i. Add a kludge like Use_cin_to_Vector mentioned earlier under VecTestL.
12. **Eigenval module**
    a. Eigenval.H and Eigenval.CPP
        i. Use Assign in place of a converting assignment statement in Eigenvalues.
    b. EigenTst No further changes.

13. **Equate2 module**
    a. `Equate2.H` and `Equate2.CPP` No further changes.
    b. `Equ2Test`
        i. Add `randomize();` to `Equ2Test.CPP`. In fact, certain initial values of this fixpoint iteration cause divergence, and the non-random version found one!
    c. `Fig9_03` No further changes.
    d. `Fig9_05`
        i. Change `y0` throughout to `y00`. Somehow some other piece of linked code has a global called `y0`!
    e. `Fig9_09` No further changes.
    f. `Fig9_10` No further changes.
    g. `Fig9_12`
        i. Change `y0` througout to `y00`. Somehow some other piece of linked code has a global called `y0`!
        ii. Remove an `int` declaration in a `for` loop. It's still in the scope of the previous one. Microsoft hasn't implemented this change in the language yet.
    h. `Fig9_16`
        i. The same puzzle occurs that happened with Borland. This won't compile, even though the same code did in `Equ2Test`. Microsoft objects to the same statement that Borland did. There's something I don't understand, but I won't troubleshoot it now.
    i. `Fig9_17`
        i. Change `y0` througout to `y00`. Somehow some other piece of linked code has a global called `y0`!