MPW C++ 3.1 Release Notes

6 September 1990

Overview

This release is Apple's first "final" release of *MPW C++.* It bears the number 3.1 because it is intended for use with *MPW 3.1.* (It is also compatible with *MPW 3.2 beta 1,* and we intend that it will also be compatible with *MPW 3.2 final* when that becomes available.) The CFront program included is called version 1.0. It is based on the final release of AT&T CFront 2.0.

New

This release of MPW C++ includes several new features (see below):

- Load/Dump (for precompiled header files)
- MultiFinder Memory Option
- MPW Shell “Mark” generation

Reporting bugs

If you find a **bug**, please report it to Apple. Use the application “Outside Bug Reporter,” found on one of the release disks. After completing the bug report, send it to:

via AppleLink: CPLUS.BUGS, TRISH, and MACDTS
(please send it to ALL 3 of the above)

or

via Internet: cplus.bugs@apple.com@internet#

or

via U.S. Mail:

Apple Computer, Inc.
Developer Technical Support MS 75-3T
20525 Mariani Ave.
Cupertino, CA 95014

System Requirements

MPW 3.1 requires a hard disk and at least 2 Mb of RAM. Actually, the MPW Shell's MultiFinder partition size comes set at 1024k. On a 1 Mb Macintosh, however, the MPW Shell only gets about 800k (not using MultiFinder), and only trivial C++ programs will successfully compile. In order to compile and link large C++ programs or programs with symbolic information, it is necessary to increase MPW Shell's memory partition size to 2048k or more or use the new '-mf' option. (To change the memory partition size, simply select the MPW Shell icon in the Finder and use the "Get Info" command from the File menu. Type in a memory partition size in the "Application Memory Size" box.) SADE requires MultiFinder and at least 2.5 Mb of RAM. MPW 3.1 requires System 6.0.2 (or later) and Finder 6.1 (or later). To run SADE and the MPW Shell to debug an MPW Tool requires at least 4.5 Mb of RAM.

Note: There are situations where the MPW C compiler runs out of stack space and crashes or corrupts the heap, when compiling code emitted by CFront. Increasing the MPW Shell's stack size to 64K seems to cure these problems. (To change the MPW Shell's stack size, use the application ResEdit to open the 'HEXA' #128 resource. This resource contains a long-word which is used to set the stack size, (e.q. the value $00010000 would set the stack size to 64K). A value of zero instructs the Shell to use its default size. NOTE: Do not set the stack size to be less than about 32K manually -- for sufficiently small values the Shell won't even be able to boot without running out of stack space.) If you run across any examples in which CFront itself runs out of stack space, please let us know.

Installation

To install from diskettes:
Start up the MPW Shell.
Insert the diskette marked **'MPW C++ Disk1'**.
Type

```
Duplicate 'MPW C++ Disk1:Install' :
Install
```

Fans of the Backup command can also use the following commands:

```
Backup -from 'MPW C++ Disk1:' -to "{MPW}" -r -a -c
Backup -from 'MPW C++ Disk2:' -to "{MPW}" -r -a -c
```

NOTE: The new C compiler will be placed in **"{MPW}"Tools:C3.2b1:C**, not in **"{MPW}"Tools:C**. We did this so that you can save away your old C compiler before installing the new one. (Just move it into the **:Tools:** directory.)

NOTE: The file CPlus.help on the disk **MPW C++ Disk1** contains help information for the CPlus and CFront tools in addition to the information contained in the **MPW.Help** file. To access this additional information, type

```
Help -f {MPW}CPlus.help CPlus
```

or simply paste the information in the CPlus.help file into your MPW.help file in the appropriate places.

Parts List

The following is a listing of the components of the *MPW C++ 3.1* release. All of the pieces in the following parts list are **required** for using the *MPW C++ 3.1* release, you cannot “mix and match” with parts from previous releases:

:Tools:

- CFront
- C (3.2b1)
- Unmangle

:Scripts:

- CPlus

:Libraries:CLibraries:

- CPlusLib.o/CPlusLib881.o
- CPlusOldStreams.o/CPlusOldStreams881.o

:Interfaces:CIncludes:

- fstream.h
- generic.h
- iomanip.h
- iostream.h
- new.h
- oldstream.h
- stdiostream.h
- stream.h
- strstream.h

:Examples:CPlusExamples:

- Count.cp
- Count.make
- CPlusShapesApp.make
- CPlusTESample.make
- Instructions
- Shapes.cp
- Shapes.h
- ShapesApp.cp

ShapesApp.h
ShapesApp.r
ShapesAppCommon.h
ShapesDocument.cp
ShapesDocument.h
StreamCounter.cp
StreamCounter.h
TApplication.cp
TApplication.h
TApplication.r
TApplicationCommon.h
TDocument.cp
TDocument.h
TECommon.h
TEDocument.cp
TEDocument.h
TESample.cp
TESample.h
TESample.r
TESampleGlue.a

Miscellaneous:

Macsbug 6.1 Unmangler
unmangle.o
CPlus.help
C++ 3.1 Release Notes *(this document)*

The C Compiler Included in This Release (3.2b1)

The *MPW 3.1* C compiler does NOT support load/dump. In order to release this feature, it was necessary for us to release a newer C compiler. It is called *3.2b1* and is needed only until *MPW 3.1* is released.

This C compiler supports some optimizations not available with MPW 3.1 C. There is good news and bad news here: the good news is that this C compiler generates better code. The bad news is that it is not as “mature” as MPW 3.1 C and it may have some strange and wonderful bugs. If you think this compiler is generating bad code, please try the “-opt off” option on the CPlus command line. This turns off all optimizations in the C compiler.

If you *really* have trouble with the *3.2b1* compiler, please notify us about it and go back to using the *3.1* C compiler. (You won't be able to use load/dump, but at least you'll have a reliable compiler.)

There are two new C options that CFront passes through to the C compiler:

```
	-opt on|off		# turn optimization on (default) or off
```

-opt off is safer than -opt on (the default). -opt on will give you faster, smaller code, but the C compiler will take longer to execute with -opt on, and it may generate incorrect code in some cases. If you write code that behaves differently depending on whether optimization is turned off or on, please notify us. This is never supposed to happen.

```
	-trace on		# generate tracing code--calls to %_EP and %_BP
	-trace off		# generate no tracing code (default)
	-trace always	# always generate tracing code--calls to %_EP and %_BP
	-trace never		# never generate tracing code (default)
```

The -trace option is most frequently used with the MacApp debugger. -trace on generates a call to %_BP at the start of each function and a call to %_EP at the end of each function. If you compile and link with -trace on, but not within MacApp, you will probably get link errors: the linker will not be able to find the %_BP and %_EP functions unless you have written your own.

Load/Dump Option

The load/dump facility can greatly reduce compile times when large header files are used. The portion of the compilation that reads the header files can be eliminated except for a few seconds to read a large disk file.

The “-dump <filename>” option saves the state of the C++ compilation (mostly the compiler symbol table) to the specified file. A matching “-load” restores the state. In effect, the compilation is suspended and resumed—the dump portion of the compile is only done once, while the load portion can be repeated for many different source files.

For example, you could say:

```
CPlus header_file.h -dump header_file.h.dump
CPlus source_one.c  -load header_file.h.dump
CPlus source_two.c  -load header_file.h.dump
```

If the dump file name ends with ':', it is suffixed with the source file name and ".dump", so the first command is equivalent to

```
CPlus header_file.h -dump :
```

The “-load” compilation can include additional headers if needed, but only one load file can read during a compile. The load occurs before any “-d” or “-u” options, so macros can be added or removed on the command line.

If you have global definitions in the header file, you will get a nontrivial "header_file.h.o" file, which you may want to include in your link. In particular, this may happen because of nontrivial "const" declarations such as “const char string[] = "abc";”.

Static objects in a header file will be compiled into the "header_file.h.o" file, where they will be inaccessible from other compiles, even one that uses "-load". Because of this, static objects cause a compilation warning.

If your headers are protected with the "#ifndef...#define...#endif" convention, you can leave your #include's in the source file since the #define symbol will be defined in the load. But you should put a #ifndef...#endif around the line containing the #include itself in the source file, so that you don't waste time reading the header files just to skip them.

To generate a compressed dump file, use “-dumpc” (“-load” is used for both compressed and uncompressed versions). This reduces the file size by about 40%, with very little change in compile speed on the Mac II or Mac SE. On machines where the CPU-to-disk speed ratio is higher than the Mac II, compressed loads may actually run faster than uncompressed. In future releases, all dumps may be compressed.

The dump file contains the state of the "-mc68881", "-elems881", "-n", "-x", and "-sym" options given when the dump file was made. When loading from this dump file, the options on the command line must match those in the dump file. If they do not, the compiler will issue a warning and exit.

The compiler will also issue a warning if -d and -u are used with -load. This is not an error, but most of the time it is not what the user intended to do: typically the user expects -d and -u to affect every file included in his program. With -load they will NOT affect the code included in the dump file. The macro values in a dump file are “frozen” at the time that dump file is made.

It is legal to have *both* the -dump and -load options on the same line. You might want to do this, for example, to load from a dump file containing all the MacApp headers, compile your own headers, and then dump the whole thing into one big dump file.

Load/Dump requires a special version of the C compiler (*MPW C 3.2b1*), included in this release.

Why did Apple add Load/Dump to C++?

The Load/Dump mechanism built into *MPW C++* is designed to counteract a common problem in large scale application development and in OOP, in general. The problem is that in order to compile one small source file (typically < 20K) the compiler must first process a much larger body of header file information (22,000 lines or 725K bytes for the MacApp and Toolbox headers). Since the information in the header files rarely changes, much of this overhead can be eliminated. Load/Dump allows you to “pre-compile” the header files and then rapidly load them into the compiler at startup. Typical speed improvements are on the order of 2–3x.

Step by Step Instructions

1. Decide which header files you want to pre-compile. Typically this will include your Application Framework (MacApp, etc.), Toolbox Interface, and Language Library header files.

2. Create a new header file called MyAppDump.h which will contain #include directives for all of header files you decided to pre-compile in step 1.

   Look through your application’s source and header files for #include references to any of the files from step 1. In each application file that contains a reference, paste a copy of the #include lines into MyAppDump.h and replace the #include lines in the source file with the following:

   ```
   #ifndef __MyAppDump__
   #include "MyAppDump.h"
   #endif
   ```

3. Bracket the body of MyAppDump.h with a #ifndef…#endif so that it looks like:

```
#ifndef __MyAppDump__
#define __MyAppDump__
...
#include <Quickdraw.h>        // to be precompiled...
...
#endif
```

4. Add a build rule to your makefile corresponding to the compile and dump of the precompiled headers:

```
MyAppDump.h.o ƒ MyAppDump.h   etc.
  CPlus MyAppDump.h -dumpc MyAppDump.h.dump ∂
    -o MyAppDump.h.o {CPlusOptions}
```

5. Modify the build rules of your application’s source files to include a “-load” of the precompiled headers, and a dependency on the dump file itself:

```
MySrc.cp.o ƒ MySrc.cp MySrc.h MyAppDump.h.o
  CPlus MySrc.cp -load MyAppDump.h.dump ∂
    -o MySrc.cp.o {CPlusOptions}
```

   (well, actually the .o corresponding to the dump — Make doesn’t yet handle dependencies for tools producing multiple output files)

6. Finally, add the object file produced by the dump phase (here MyAppDump.h.o) to your link dependency and link list since it may contain definitions of global const objects, vtables, bodies of inline virtual functions, etc.

Before You Report a Load/Dump Bug...

Because of the way this release is packaged, a lot of you might see the C compiler crash in the following way when you try load/dump:

```
CPlus "{CIncludes}"stdio.h -dump stdio.dump
# C - Fatal Error : 430
# unexpected token in the token input file
#-------------------------------------------------------------
	File "HD:MPW:Interfaces:CIncludes:stdio.h"; Find •!0:§!1; Open "{Target}"
#-------------------------------------------------------------
# C - Aborted !
```

This is probably because you are using an old C compiler that does not support load/dump. The C compiler sees the tokens that CFront sends it for load/dump and does not understand them. Get the C compiler installed in **"{MPW}"Tools:C3.2b1:C** and put it in **"{MPW}"Tools:C**. The 3.2b1 C compiler understands the load/dump tokens.

The above example is also a good test for load/dump. If you cannot make a dump file out of stdio.h, then you have something very wrong with your system.

Multifinder Memory Option

When the “-mf” option is used, MPW C++ will request MultiFinder memory when the MPW shell partition is not large enough to complete the compilation. Of course, the amount of MultiFinder memory available depends on what other applications are running; and C++ use of this memory will reduce the amount of memory available for running other applications.

MPW C++ releases MultiFinder memory at the end of each compilation, even when interrupted with Command-period. However, after certain catastrophic errors (such as a bus error recovered via "g stoptool" in Macsbug), MultiFinder memory may not be released. If this occurs, you must quit the MPW shell to free the MultiFinder memory.

With the -mf option, you can keep the MPW shell partition small enough (e.g. 1-2 Mb) to run other applications with MPW, and still be able to do large compilations without sizing up the MPW shell.

The MPW Linker ("Link") and Lib tool also accept the -mf option and we recommend you use it whenever possible.

The CPlusShapesApp example has been modified to use -mf. The CPlusTESample example has not. You can compare the two to see the difference.

Generating Marks

The “-mark” option tells MPW C++ to install MPW shell “marks” into the source files(s). Note that only the file(s) listed on the CPlus command line are marked; include files are unaffected. (Include files are read so often that marking them each time would significantly degrade performance.)

Include files can be marked by compiling them directly:

```
CPlus -mark all something.h
```

You can mark functions, types (including classes), and global data items. The “-mark” option is followed by a list of one or more sub-options [fcts,types,data,all]; the syntax is similar to the -sym option.

The added C++ marks begin with “—” (option-underscore). Any existing marks beginning with “—” are removed, and the new marks are merged with the old non-C++ marks. The marks you added are thereby preserved (unless they start with “—”).

Since the marks are maintained in the resource fork of the source file, there is no danger of corrupting the text. The modification time of the file is not changed by the -mark option. (Actually, it’s reset to its original value.) Marking does not work with source files containing “#line” directives.

Pragmas

Unrecognized pragmas (for now, all but “#pragma segment”) are now passed through to the C compiler. However, any pragmas encountered within a function body, class body or other file-level definition are emitted before the definition that contains them. Since there is not an exact one-to-one mapping of C++ constructs to C constructs, position-dependent pragmas may not produce the expected effect. In particular, C type declarations used to implement C++ types may be emitted out of order or not emitted at all, if not needed.

Unmangling Tools

The **Unmangle** tool is an *MPW* tool which attempts to “unmangle” the name passed to it on the command line. (CFront appends a “type signature” to the names defined by the user; such a name is called a “mangled” name. The type signature is necessary in order to distinguish different overloaded functions. The linker cannot understand overloading, so CFront resolves it by making distinct names the same way a user would in a language like C that does not allow overloading.)

The command

```
unmangle mangled_name
```

yields the unmangled name, or reports that the name was not a mangled one. This can be useful when deciphering error messages from the linker about unreferenced externals with mangled names.

NOTE: This tool, and the library described below, are not identical to the AT&T unmangler. For example the unmangle tool and library do not unmangle local variable names.

The file “Macsbug 6.1 Unmangler” on the disk “MPW C++ Disk2” contains a resource that should be pasted into your MacsBug 6.1 “Debugger Prefs” file with ResEdit. This resource allows MacsBug 6.1 to unmangle the function names produced by C++. Thus, instead of seeing mysterious, and potentially unsettling, names like

```
        __ct__8TPN_ExprFPcN21
```

in the disassembled code, or in the “pop-up name selector window” (type command-colon) you’ll see

```
        TPN_Expr::TPN_Expr(char*,char*,char*)
```

instead. CFront/C generates these MacsBug names by default or with the command line option “-mbg full”.

(A translation by hand of the above name: “__ct” means “constructor”. “__8TPN_Expr” means this is a member function of “TPN_Expr”. 8 is the number of characters in “TPN_Expr”. “F” means this is a function as opposed to a static data field or some other kind of object. “Pc” means “pointer to char”. “N21” means 2 parameters of the same type as the 1st parameter, i.e. repeat “char *” twice. The unmanglers do this translation for you. If you need a detailed explanation of this encoding, see Bjarne Stroustrup's paper, *Type-Safe Linkage for C++.*)

We have also included the unmangler in library form in the file unmangle.o. The function unmangle(), which decodes C++ mangled symbols (i.e. a symbol with a type signature), behaves in the following manner:

```
        int unmangle(char *dest, char *src, int count);
```

where:

```
        dest  = pointer to result buffer
        src   = pointer to mangled symbol
        count = max chars to write not including null
                    (e.g. sizeof(buffer) - 1)
```

The return codes are:

```
        -1 = error; probably because symbol was not mangled,
                    but looked like it was
         0 = symbol wasn't mangled; not copied either
         1 = symbol was mangled; unmangled result fit in
                    buffer
         2 = symbol was mangled; unmangled result truncated
                    to fit in buffer (null written)
```

Grunge-Level Details

Here are some interesting “bits-o-trivia” you may need to know:

• Some C++ users may be curious about where and when CFront chooses to emit virtual tables. Normally you don't have to worry about virtual tables (*vtables*), which are generated to support virtual functions. MPW C++ usually generates the tables in just one object file (out of all the files that use the class), and the link will work correctly. There should be no duplicate or missing vtables. (Vtable names begin with "__ptbl" and "__vtbl".)

You should be careful to recompile all source files which reference a class when you change the class, even if you think the change does not affect some files. If you don't recompile the file that happens to generate the vtables for the class, the tables will not be updated and your program may not run correctly.

Sometimes the vtables for a class are generated from all source files that use the class. When this happens, you will see a "duplicate symbol" warning from link. You can safely ignore this warning. You can also fine-tune the vtable generation using the "-vtbl0" and "-vtbl1" options. For example, you can create a file which includes all of your class declarations, then compile that file with "-vtbl1" and all others with "-vtbl0".

In some rare cases, e.g. when a virtual function is written in assembly code, you may need to know how CFront decides which file to put the vtables into. The exact criteria are subject to change, but the current method is given below. (This description is only approximate.)

In general, CFront tries to find a "key" function in the class. For regular classes (not derived from PascalObject), this function is the first non-inline virtual function in the class definition. (A function is considered inline if the body is given in the class, or if the definition of the function uses the "inline" keyword.) If there is a key function, then virtual tables will be generated for a file only if that function is defined (has a body) in the file. Since only one file should define the key function, only one copy of the vtables should be produced. If there is no key function, vtables are always generated. This may lead to duplicate vtables and warnings from the linker.

Classes derived from PascalObject use a similar method to determine when to emit Pascal method tables. However, the criteria for selecting the key function are too complex to describe here.

• In order to handle static constructors and destructors in C++ code, CFront and the Linker conspire to create a special segment named “%_Static_Constructor_Destructor_Pointers”. This segment is checked as the application starts up to see if any static data initialization/cleanup is required. Removing this segment or changing its name will cause this process to fail.

Recent Changes and Bug Fixes

The following is a summary of the bugs fixed since the B1 release of *MPW C++* (based on AT&T 2.0 Final). First time users of *MPW C++* will also be interested in this list since most of these bugs are present in the AT&T 2.0 Final software, but have been fixed in subsequent versions of Apple’s *MPW C++*.

Bugs fixed in the final release:

• Overloaded operators in nested classes caused the load/dump mechanism to get a bus-error.

Bugs fixed in the B4 release:

• The code generated to access members of virtual base classes is now correct. This "known bug" had been so bad in previous versions that it was unsafe to use virtual base classes at all. It should now be safe.

• The name of the compilation unit was given to C and SADE incorrectly and resulted in bizarre warnings from C -- for example, “Can't find modification date on file.” The name given was garbage due to an uninitialized data structure.

• “Volatile” was removed from the grammar. It is still a keyword. Therefore it is not legal to define an identifier called “volatile” and anywhere you use the keyword “volatile”, you now get a syntax error. Formerly, CFront would silently generate incorrect C code for it. “Volatile” is not implemented in the 3.1 B4 release and until it is it will not be legal in the grammar. This way users will not rely on it when it does not do what they expect it to.

Bugs fixed in the B2/B3 releases:

• Although the C++ Reference manual states (p. 3) that all identifiers containing a double underscore are reserved, such names are widely used (including in the C++ MacApp headers). Although CFront accepts these names, they can cause problems ranging from bad code to compiler bus errors. As a partial solution to this problem, MPW C++ version B1 emitted warnings when identifiers most likely to cause problems were used.

In this release such identifiers are handled correctly. However, double underscores should still be used sparingly, since you may inadvertently write an identifier which duplicates a name invented by CFront (for example, temporary variable names). If this happens, you will get a C compilation error because of the duplicate identifier.

• Referencing a type name as if it were a member function could cause a compile-time bus error.

• Initializing a local static structure with another structure caused a compile-time bus error.

• Incorrect virtual tables were generated when a visibility declaration was used. This could cause a run-time bus error.

• In a void function, a return statement with a return value of type void (as opposed to no return value) was not flagged with a warning.

• Comparison of a “pointer to member function” variable to a constant “pointer to member function” generated bad intermediate C code, causing the C compiler to emit errors.

• Comparison of two “pointers to member functions” using the “!=” operator caused incorrect C code, so result of the compare was sometimes wrong.

• When using an overloaded “operator new” with extra arguments, at a point where memory is nearly exhausted: if a “new” using extra arguments failed, and a regular (no extra arguments) version was provided, the regular version was called; this was wrong. For example, if the first call failed and the second call succeeded (because it allocated less memory), the object would be smaller than expected.

• Extra commas in the middle or at the end of an enumerator list (in an “enum” declaration) were not flagged as a syntax error. (One extra comma at the end is allowed.)

• Operator conversion functions could not be pure virtual.

• Default arguments in the (base/member) initialization list of a constructor sometimes caused a bogus “sorry, not implemented” error.

• When a constructor used a pointer declared to point to an abstract class (but really pointing to a derived class), and called a member function which was pure virtual in the abstract class, a bogus compilation error was emitted. The error should only be emitted when the pointer is “this”, since calls using “this” in a constructor are non-virtual.

• When a pointer of the form “&ptr” (where “ptr” was a class object) was cast to point to a virtual base class, CFront aborted with an internal error.

• A bogus warning was emitted for certain “switch” statement where the switch expression was an enumeration. CFront warned about possible missing cases when there were almost, but not quite, as many cases as enumeration values. This warning should not have been emitted when there was a default case.

<u>Macintosh-specific Bugs</u>

• Conversion operator member functions could not be implemented as trap functions; now they can.

• Destructors for PascalObject classes referenced undefined variables of the form “__ptbl...”.

Bugs fixed in the B1 release:

• When local classes were used, certain virtual tables and functions were not emitted in the intermediate C code. This resulted in undefined symbols in the link. (A local class is a class, struct or union where the class type itself is declared within a function. Declaration of an object of class type is NOT a local class.)

• Names beginning with two underscores and an uppercase letter (__[A-Z]) are reserved (AT&T Product Reference Manual §2.4). Use of such names can cause CFront to generate incorrect code and/or crash. In this release, these names are now flagged with a warning. Similarly, class names should not contain a double underscore (no warning is produced in this case).

There may be other cases where a double underscore causes a problem in the CFront or C phases of the compilation. In this release, names containing a double underscore should be avoided if at all possible. (Note that §2.4 of the Language Reference Manual states that "identifiers containing a double underscore are reserved for use by C++ implementations and standard libraries, and should be avoided by users.")

• The *MPW 3.1 B* versions of **CLib881.o** and **CType.h** are now included with *MPW C++ 3.1 B1*, as the spelling of one of the internal data structures was changed, and could cause unresolved reference errors at link time, for *MPW 3.0* users.

Known Bugs and Inconsistencies

• There are still a few areas where CFront is either badly broken or seriously wounded. These areas include:

Inline Functions -- there are many limitations concerning what is permissible within an inline function.

Inefficient Code Generation -- most cases of inefficient code result from the fact that semantic information is lost in the translation to C. This should be viewed as evidence that C is a poor choice of intermediate languages, rather than that C++ inherently produces poor code.

In addition to these general areas, there are still many minor inconsistencies between C++ as described in the AT&T Language reference manual and C++ as implemented by CFront 2.0. Remember, C++ is a young language and CFront is an inherently imperfect implementation of the language.

• Include files are searched for in the <u>current directory</u> first, not in the directory containing the source file.

• *MPW C++* does not allow “incomplete types” to appear in classes, structs or unions (neither does ANSI C). This may cause some difficulty for MPW C users who are able to write:

```
struct T {
        int a;
        int intarray[];          /* incomplete type! */
};
```

to describe data types that end with a variable length array.

• *MPW C++* currently “narrows” floating point return values in functions whereas *MPW C* leaves the return value as an extended value. This inconsistency may cause some difficulty when mixing functions written in C and C++ that return non-extended floating point values. We anticipate that future releases of *MPW C++* will follow the convention used by *MPW C*.

• **PascalObject** Limitations and Peculiarities

1. When using object hierarchies based on the built-in class **PascalObject**, the file containing the main entry point, i.e. “main()”, must contain at least one class derived from **PascalObject**, in order for the **PascalObject** initialization code to be called. Something like the following will suffice in cases where no other references would appear.

```
class foo : PascalObject {};
```

Note that this precludes writing your main entry point in C. Pascal or Assembler may be used **if** a reference to an Object Pascal object is similarly referenced.

2. Pure virtual functions are not allowed in **PascalObject** hierarchies. Currently, CFront will accept the pure virtual declaration syntax, but the linker will fail with an unimplemented function error.

3. Pointer-to-members in **PascalObject** hierarchies are not supported due to implementation differences between C++ and Object Pascal. (CFront currently does not complain about creating or using such a pointer-to-member, but the resulting application will either fail to link properly or will fail at run-time with an error in the method dispatcher.)

4. The **pascal** keyword is broken in the specific situation where one attempts to call a C function which returns a pointer to a Pascal-style function. The C compiler currently misinterprets the C function as a Pascal-style function and the function result is lost.

5. The *MPW C++* keyword **inherited** can only be used within **PascalObject** hierarchies. This is a design decision, not a bug (We are trying to avoid gratuitous changes to C++, except where necessary for compatibility with the Macintosh, here specifically MacApp.)

• Incorrect code generation:

1. Using pointers to member functions of a class as default parameters to member functions of the same class sometimes causes problems. Using a member function as a default argument to a subsequent member works:

```
struct A {
        void foo();
        void bar(void (A::*pfoo)() = &A::foo);
};
```

However, using a member function as a default argument to a previous member does not work:

```
struct A {
        void bar(void (A::*pfoo)() = &A::foo);
        void foo();
};
```

A workaround does exist, however, for this case:

```
struct A;

extern void (A::*pfoo)();

struct A {
        void bar(void (A::*pf)() = pfoo);
        void foo();
};

void (A::*pfoo)() = &A::foo;
```

You could also use the Apple extension that a variable can be declared "extern" and then "static", so the last line would become:

```
static void (A::*pfoo)() = &A::foo;
```

2. Statements of the form:

A a[2] = {A(*opt_parms*), A(*opt_parms*)};

where A is a class, fail, since the constructors for the objects being assigned into the array are never called.

3. Statements like:

```
int a;
int b;
a = b = 0;
```

still produce the erroneous “b used before set” warning.

4. There is a problem with some forms of inline operators. If an inline is declared to return a value and a path through the function does not, CFront may expand the function to just return “0”. It may not warn you about this.

For example:

```
inline int operator --(stack& s) {
        if (size > 0) return *top;
        else
                cout << “Error”;
}
```

The path producing an error does not explicitly return and CFront may simply generate an expression that evaluated to 0 in that case.

5. In the following example, CFront can coerce a B* to an A* for f(), but cannot coerce a B** to an A** for ff().

```
struct A { };
struct B : A { };
```

```
void f(A*);
void ff(A**);

B *bp, **bh;

void g()
{
	f(bp);
	ff(bh);
}
```

6. In the following example, CFront incorrectly reports that it cannot access operator new() from the private base class Base.

```
class Base : HandleObject {};
class Derived : private Base { };

main()
{
	Derived *dp = new Derived;
}
```

This problem seems to be limited to classes derived from HandleObject. As a workaround, override operator new in the Derived class to call HandleObject::operator new.