NOTE: This Technical Note has been retired. Please see the Technical Notes page for current documentation.

Technical Note PT555

MPW C++ Q&As

CONTENTS

This Technical Note contains a collection of archived Q&As relating to a specific topic - questions sent the Developer Support Center (DSC) along with answers from the DSC engineers. Current Q&A's can be found on the Macintosh Technical Q&A's web site.

[Oct 01 1990]

Labels in block with destructors not implemented

Date Written: 3/2/93

Last reviewed: 7/2/93

When I try to compile some C++ code that I'm porting from another platform, I get the error message shown below. Can you shed some light on what the error means? I can avoid it by removing labels, but sometimes it's difficult to fix. What's the real cure?

```
# 4:14:35 PM ----- Executing build commands.
    CPlus -s tops1 -model far -sym on -mf test.cp
    Set Echo 0
```

The error message is correct; labels in a scope that locally defines objects with destructors aren't supported. The most appropriate fix would be for AT&T to fix their `CFront` code to support this, but that's not going to happen right away.

The variety of twisted code paths that become possible when labels and gotos are used probably gave the compiler programmers conniptions. When code can contain constructors and destructors, the C++ compiler has to be extremely careful to construct each object only once and destruct each object only once. This probably was a problem for the C++ compiler if gotos were allowed, so the compiler programmers must have chickened out and didn't write the necessary code (at least, not yet). The only solution to this error is either to avoid the label and goto constructs (which you've been trying to do) or to keep any objects with destructors out of the block that contains the label.

C++ stand-alone code and memory management

Date Written: 9/4/92

Last reviewed: 11/1/92

We're trying to use MPW C++ 3.2 in stand-alone code using the technique described in "Polymorphic Code Resources in C++," in *develop* issue #4. However, we've found that `new` and `malloc` both access A5. Is this correct? What's the workaround?

Unfortunately, you are absolutely correct, in that both `new` and `malloc` attempt to use A5-relative globals. Malloc uses global data space to maintain structures that track its use of memory (the free list, for example); new, because it is based on `malloc`, suffers from the same limitation. This will limit your use of `malloc` and `new` to when you have an A5 world installed. Also, you will need to be very careful with your memory allocation, because `new` and `malloc` do not generally dispose of the memory they request from the Macintosh. They allocate Macintosh Memory Manager blocks with `NewPtr`, then use that memory to satisfy requests made by the program. When all of the memory is freed, the `NewPtr` blocks are not disposed of - they will remain to clutter up the heap. This is particularly a problem with stand-alone code, because most stand-alone code runs as a parasite in someone else's heap; this means that there may be no simple way to deallocate these blocks; they may remain just to clutter the host's heap.

Our only suggestion is to discontinue use of the ANSI library memory calls; instead, use the Macintosh memory calls, `NewPtr` and `NewHandle`. These do not require the use of global data space and when their memory is freed, it gets deallocated. You can change your calls to malloc to use `NewPtr`, and you can override new to allocate its space with `NewPtr`.

MPW C++ pragma support limitations

Date Written: 1/15/92

Last reviewed: 6/14/93

When used from MPW C++, `pragma unused`, `pragma force_active`, and `pragma` once don't appear to work. In fact, pragma unused actually causes a C compile-time error. Why does this occur in spite of assurances in release notes that all pragmas are passed on to the C compiler?

The problem with pragmas and C++ is that the `CFront` compiler generates C code, and during this phase it also shuffles around the source code lines, so the pragma doesn't end up in the same place as originally intended. Also, `CFront` moves any pragmas inside the function body outside, because it can't do much with the pragmas, and the best bet is to move them just outside the body for the C compiler. This means that any pragmas stated inside the function body are unusable in real

life.

Here's a summary of how pragmas work with C++:

- `pragma segment`, `pragma parameter`, and `pragma processor` should work OK.
- `pragma force_active` may or may not work, depending on the code case.
- `pragma warnings` and `pragma pop/push` should work in most cases, depending on the code movement.
- `pragma trace` should also work, especially if it's defined just before a function or member function.
- `pragma unused` and `pragma` once won't work, alas.

For more information about `pragmas` and C++, please consult the MPW 3.2 C++ documentation:

Undefined Pascal class constructor and Unmangle tool

Date Written: 6/22/90

Last reviewed: 6/14/93

When a constructor is declared in a Pascal object base class (used, for instance, in MacApp) and not defined, the error message contains a garbled name (strange characters at the beginning and end). The Unmangle tool is in my Tools folder. Do you know what the problem is?

The Unmangle tool doesn't get invoked automagically; you'll have to use it to unmangle the symbol that's given in the error message yourself.

MPW C++ 3.1 and volatile variables

Date Written: 7/20/90

Last reviewed: 8/1/92

The volatile keyword isn't supported by `CFront` 2.0 (upon which MPW C++ is based). Until it is, you can keep a variable from being assigned to a register by taking its address (using the & operator).

To keep your code readable (and to make it easier to make the appropriate fixes when volatile works), you might use the following macro (MacApp 3.0 and the System 7 Finder both do this!):

Whenever you define a variable to be used in handling a failure, invoke this macro after you define it as follows:

This workaround isn't free: it has the side effect of adding a harmless (but useless) LEA instruction to the resulting object code.

Use extern to link C++ code with object code written in C

Date Written: 12/12/91

Last reviewed: 6/14/93

I'm having trouble linking a program that is written in C++ with object code written in C. The example below illustrates the problem. The main program, compiled with CPlus, doesn't find the routine, foo(). The error message is:

```
# 3:27:27 PM ----- Build of test.
# 3:27:27 PM ----- Analyzing dependencies.
# 3:27:28 PM ----- Executing build commands.
  CPlus a.cp -o a.cp.o
   Set Echo 0
  C b.c -o b.c.o
  Link -d -c '????' -t APPL a.cp.o b.c.o "splinter:Open
Interface:MPW:Libraries:CLibraries:"StdClib.o "splinter:Open
Interface:MPW:Libraries:Libraries:"Runtime.o "splinter:Open
Interface:MPW:Libraries:Libraries:"Interface.o -o test
### Link: Error: Undefined entry, name: (Error 28) "foo__Fv"
```

This is actually a common stumbling block, but fortunately one that is easy to address. In a nutshell, you need to declare your routines to have external linkage by declaring them using extern. For a good description of extern versus static declarations (and how this affects external versus internal linkage), please see page 35 of the *AT&T C++ Language System Reference Manual* (Release 2.0). This comes with MPW C++ 3.1. You'll probably also find this explained in most good C++ books.

In the MPW 3.2 C interface files, for example, you'll find routines defined as follows (basically the extern "C" is declaration is included if you're compiling with C++):

```
#ifdef __cplusplus
extern "C" {
#endif

pascal OSErr AECreateDesc(DescType typeCode,
            Ptr dataPtr,
            Size dataSize,
            AEDesc *result)
  = {0x303C,0x0825,0xA816};

#ifdef __cplusplus
}
```

Here's an additional excerpt on the subject, from a paper by Bjarne Stroustrup:

C Linkage

This leaves us the problem of how to call a C function or a C++ function "masquerading" as a C function. To do this a programmer must state that a function has C linkage. Otherwise, a function is assumed to be a C++ function and its name is encoded. To express this, an extension of the extern declaration is introduced into C++:

```
extern "C" {
      double sqrt (double);    // sqrt(double) has C linkage
```

This linkage specification does not affect the semantics of the program using `sqrt()` but simply tells the compiler to use the C naming conventions for the name used for `sqrt()` in the object code. This means that the name of this `sqrt()` is "sqrt" or "_sqrt" or whatever is required by the C linkage conventions on a given system.

Bottom up C++ construction, base classes through derived class

Date Written: 10/23/90

Last reviewed: 8/1/92

When calling overridden functions from the constructor of a superclass, the functions in the superclass are called. Using the MPW C++ TE example, when a function such as `StackNeeded` is called by the superclass constructor of an application object, the superclass function by that name is called when one would expect an overriding function returning the derived class's needed value would be called instead.

To work around the problem, I created a function in the superclass called by the subclass's constructor that is not a constructor (called it `ReluctantInit`). I would rather use the constructor, however, instead of the derived class having to "know" to call an INIT function in the superclass. (You might want to fix `TECPlusSample`, which makes this very same mistake.)

According to the AT&T docs, the superclass's constructor should be called before the derived class's. The syntax SubClass::SubClass:SuperClass (used in the derived class constructor) therefore does not make sense to me, as it suggests that the derived class constructor is called first.

I have bad news: You've uncovered a weirdness (sort of) in how MPW C++ constructs things. Apparently C++ constructs things "from the bottom up," meaning from base classes through derived classes. Therefore, in a base-class constructor, calling a member function will call the base-class member function rather than the derived-class member function, because the derived class hasn't been constructed yet!

Your workaround, although I agree that it's philosophically displeasing, is the only one that I can think of offhand.

Where to find documentation on virtual table generation

Date Written: 11/5/90

Last reviewed: 8/1/92

Using the MPW C++ -I option when linking some of our source files, we noticed the virtual table sizes in the output file are different for different objects. This we expected, but we couldn't discern any pattern. What comprises the vtable?

The topic concerning generation of vtables is very big. As you know our MPW 3.1 C++ compiler originates from AT&T CFront 2.0, so we generate vtables in the same manner as the original C++ compiler.

You can find information about virtual tables and how the tables are generated/accessed from the following sources:

"MPW C++ 3.1 Release Notes"

Macintosh Technical Note "Multiple Inheritance and Handle Objects"

"Multiple Inheritance for C++," included with MPW C++

Chapter 10 of *Annotated C++ Reference Guide* by Ellis and Stroustrup

How to use C++ dump/load capabilities

Date Written: 12/10/90

Last reviewed: 8/1/92

It seems like this dump facility is one of the features that everyone takes for granted, like me, and in real life it's more tricky than the examples given in the manual.

I will try to write down what I learned experimenting with C++ dump/load and especially MacApp.

A. To start with, the best thing to check if you have the right C compiler for dump/load is to do a simple test under Macintosh Programmer's Workshop (MPW). You need to have the C compiler included with the MPW C++ 1.0 final, and that one is MPW C 3.2ß1 (3.2ß1 or later). If you have a C compiler from the MPW 3.1 release you won't get dump/load to work at all.

OK, do a

or something similar. If the C compiler complains about "unexpected tokens" you have the wrong C compiler. This is especially true if you have one of the first APDA ones, the MPW 3.1ß1 release.

B. If this worked, the next step is to get it working with a generic C++ program. The example given in the C++ release notes (C++ 1.0 final) is a good one that should work. Use the rules at page 6-7 to build a single uniform header file that will be used to include the other header files.

C. Let's now try with MacApp. Here we have some really interesting issues. To start with, MacApp 2.0 has a flag for MABuild that takes care of the whole dump/load management concerning the MacApp C++ header files. If you include a `-CPlusLoad` to MABuild it will happen automagically. The dumped file is stored into a folder under the MPW folder called Load Files. In the MacApp folder there's a file called Startup where you can modify where the dump file is stored, as well as if the `-CplusLoad` is done every time without the flag to MABuild.

So in this case the dump/load management is handled for you. The drawback is that you could not include your own header files. Which leads to:

D.1. How to include your own header files. Here I actually found two different ways to do it. The first one is to hack a file under the MacApp:Tools folder called Build Rules and Dependencies. If you search this one you will find a place where the dump/load is defined. Comment out the line that says, "Include you own files here." I tested this and it worked OK.

D.2. Now let us assume that you want to build your own MAMake file. This is also OK. The trick in this case is to carefully specify where the dump, source, and object files reside by using the prefix {ObjApp} and {SrcApp} for each file. The other trick is to have a dummy object file as one of the dependency rules that builds the dump file in the first place. It is important that the OtherLinkFiles variable has this dummy file in its definition!

The dump/load is about 2-3 times faster during the C compilation phase due to the fast load of static information. An example of an MAMake file for MacApp ("DemoText with C++ load/dump") is on AppleLink.

MPW C++ compiler external link bug and workaround

Date Written: 1/8/91

Last reviewed: 8/1/92

Is there a bug in the dump/load functions of the Macintosh MPW C++ compiler (CFront) on declarations of external routines with C linkage--for example, the GraphAccel.h file?

You are absolutely right, it won't work. As a workaround you might try something like:

```
// This one works...
extern "C"     { void SwapHandleBytes(Handle theBuffer);
    }

// This one won't...
```

In other words, put braces around the external link statement, and it works OK with the dump make rules. This is the reason most of the header files in CInclude files work without problems with dumping.

Function pointer won't return type pointer in union or struct

Date Written: 1/25/91

Last reviewed: 8/1/92

Why is it that I cannot define a Macintosh function point that returns a pointer to a type (such as a long) in either a union or a struct, though I can outside of one. In fact, I can do this in C, but not in C++. The error occurs when compiled by C++. When compiled only in C, it compiles with no errors.

```
----------------------(test.c)------------------------
pascal long* (*biz) ();
union waka
    { long* (*baz) (); pascal long* (*foo) (); pascal short (*bar) (); };
```

Is this a bug in Apple's MPW C++ implementation?

It seems like this is a real bug (that is, any Pascal declared function pointer inside a union/struct that tries to return a pointer to a val won't compile). I also tried to typecast this, but it naturally failed (because the typecast maps back to the original declaration).

I have to report this as a bug to C++ engineering. Meanwhile, the only workaround is either to use a nonPascal function for the work, or to return nonpointers. Note that there's a known bug with Pascal functions and MPW C++: The Pascal keyword is broken in the specific situation where one attempts to call a C function that 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.

Workaround for MPW C++ -w2 option limitation

Date Written: 1/29/91

Last reviewed: 8/1/92

If I compile with the -w2 option, `CFront` generates an error message for the anachronism "overload" instead of a warning message. Why is it generating an error for this instead of a warning?

It seems that the -w2 option is a little bit too rough with good old "overload" keywords. ARM (page 405) also defines that overload could still be used in function declaration (even if this is not needed, and in the long run it's gone, maybe ANSI C++ will not include this at all).

Anyway the -w1 option did not scream. Also if you have a lot of old C++ code with billions of overload declarations, do a "#define overload " so those will not be used with -w2 testing. Meanwhile I will send a small bug report to the C++ dudes.

X-Ref:

Annotated C++ Reference Manual by Ellis & Stroustup

Put MPW C++ inlines in class definition header file

Date Written: 6/19/91

Last reviewed: 6/14/93

According to the AT&T Reference Manual, it's possible to declare a constructor as inline, but in MPW C++ it only works if the definition of the constructor is in the declaration. For example, declaring the constructor of `TEDocument`

(TESample) as inline results in a link error because the symbol table of the linker is wrong. Is there a solution?

Inline member functions must be in the same header file (xxx.h) as the class definition. If you do this with the inline constructor, the linker will not complain. Also, CFront tries to automatically inline most of the constructors by default.

Workaround for Revert failure with multiple forks open

Date Written: 9/17/91

Last reviewed: 6/14/93

Our application, written using C++ and MacApp, works under System 6, but, the Revert mechanism fails under System 7.0. We use both forks of our document's file and keep both open. The failure occurs way down in MAOpenFile with a -49 error. It appears that the problem is related to the fact that ReadFromFile calls OpenAFile, even if both forks of the file being reverted are already open. I've overridden this behavior and things seem to work. Am I going to get into any additional trouble by doing this?

Some access permission strategies have changed between 6.0.x and 7.0. Until these changes are incorporated into MacApp, the only solution, as you found out, is to tweak the ReadFromFile-OpenAFile-MAOpenFile sequence so it doesn't try to reopen an already opened fork. Seems like the old Revert code broke with the new System 7 behavior.

Test your fix also with an AppleShare-based file, checking for -54 permission errors and errors inside PBHOpenDeny.

Script for filtering out C++ dump/load warning

Date Written: 10/4/91

Last reviewed: 6/14/93

Is there a way to turn off the warning message given by C++ when using the -load option? The message, "warning: -d or -u option(s) will not affect code saved in dump file," makes it hard to spot real errors in our build log.

The warning about -d and -u options will be bounced to the -w1 flag in a future C++ release. Meanwhile, here's a script for filtering unneeded warnings, based on streamedit (MPW 3.2). Note that because of the way pipes are implemented in MPW (files), the streamedit script that prints out the filtered information is not triggered until the compiler/linker and so on is finished at the left side of the pipe. This can be used for filtering any kinds of warnings that the developer would consider not to be important (even if all compiler and linker messages should be considered important information).

```
MABuild -nodebug -sym DemoText [[Sigma]] Dev:stdout | [[partialdiff]]
```

MPW 3.2 C objfilegen bug requires simpler statements

Date Written: 8/12/91

Last reviewed: 6/14/93

I'm using MPW C++ streams, and sometimes the compiler seems to generate bad code when I build a large statement, with lots << of << leftshift << operators. Why, and what can I do about it?

This is a known bug in the MPW 3.2 C compiler, but we don't know when it will be fixed. It turns out that CFront translates your complex statement into a nested series of procedure calls. Eventually, this series becomes too complicated for the C compiler to deal with. Unfortunately, the C compiler doesn't detect this condition itself. The only workaround is to make simpler statements (try to keep it to a dozen <<'s or less).

MPW C++ 3.2 is based on AT&T USL CFront 2.1

Date Written: 3/24/92

Last reviewed: 8/1/92

Do you have a CFront that supports AT&T's version 2.1?

MPW C++ 3.2 is based on AT&T USL CFront 2.1. Apple recommends it for all development, except for MacApp 2.0.x development. Please note that you must use MPW C++ 3.2 if you are using MacApp 3.0.

MPW C++ 3.2 is available from APDA and is also included on the E.T.O. CD as of E.T.O. #7. The specific path for MPW C++ 3.2 on the CD is as follows: "E.T.O. #7:Tools - Objects:MPW C++:MPW C++ 3.2". In the same directory, you will find a readMe TeachText document with more information about this particular release of C++.

Creating C++ objects from resources

Date Written: 2/21/91

Last reviewed: 8/1/92

Can you give me an example of how to create C++ object from resources in the way NewTemplateWindow creates Object Pascal objects from view templates?

The scheme for creating C++ objects from resources is the same if you use PascalObjects as the base class as it is with Object Pascal. Check the method NewObjectByName in MacApp 2.0, and it should be fairly easy to transpose that to C++, if you really want to do this.

Concerning non-MacApp base class objects, unfortunately you are on your own. There are many ways to obtain persistent objects in one form or another. For instance there's an example about this in *C++ programming with MacApp* by Wilson, Rosenstein, and Shafer, called TStream. Even if this class is rudimentary it should show some of the guidelines, and the new MacApp 3.0 release includes TStream.

Another article about metaclass handling is in the *MADA Newsletter,* Volume 3, No 5, "Meta-information in MacApp 2.0" by Larry Rosenstein.

MPW C++ "free store exhausted" error

Date Written: 3/8/91

Last reviewed: 8/1/92

I can't compile any of the MacApp C++ example programs without getting a CFront "free store exhausted" error from CFront. I cannot find any reference to this diagnostic in my MPW documentation. The example C programs compile OK.

Because the MPW C++ compiler is a port of the AT&T CFront, which was designed with the assumption that all computers in the world have virtual memory paging (spell UNIX), it makes notorious use of memory. It doesn't care about the free memory because it assumes that unused memory will be paged out temporarily and new free pages will be available for the compiler. Enter PC and Macintosh systems, with no virtual memory paging. This is the reason the MPW C++ needs a lot of temporary memory.

Fortunately the peak "need memory" points are infrequent, mostly during the parsing phase. A switch called "-mf" can be used with all the MPW compiler, lib, and link tools. It uses the MultiFinder temporary memory, which is not in use normally and is sort of a pool of unused memory.

If you include the -mf switch with the set of `CFront` options, either at the MPW Shell level, in the Makefile, with `MAMake` using the `-CFront` -mf, or even defining it in the Startup file inside the MacApp folder as one of the default CFront parameters, then you should have a far easier time compiling MacApp/C++ code.

If this does not help, increase the MPW Shell application heap size as well (Get Info on the MPW Shell icon in the Finder). The shell size should be at least 2.5 MB, or up to 3 or 4 MB if possible. If you have even more memory and you use MPW most of the time, increase the size to near the top of the memory range.

(UNIX is a registered trademark of UNIX System Laboratories)

Intro to OOP tutorial Lab 1 interface file update

Date Written: 6/17/91

Last reviewed: 8/1/92

When I try to compile Lab 1 of the Intro to OOP CD-ROM tutorial, I get a stack smashed into the heap error! Any ideas?

Try increasing memory partition for the MPW Shell using Get Info. Also, the MPW interface file TYPES.h has been modified since MPW 3.1. You will need to edit one line of the file, UTILS.h. UTILS.h can be found in the folder: "IntroToOOPLabs :C++:CourseLibrary:" Here is the segment of code as it appears in the shipping version of intro to OOP:

```
inline void Copystr255(const Str255 from, Str255 to)
{
     BlockMove((Ptr) from, (Ptr) to, Length(from) +1;
```

Replace the above code with:

```
inline void Copystr255(const Str255 from, Str255 to)
{
```

Resource fork handling with streams support

Date Written: 6/1/92

Last reviewed: 6/14/93

Where can I find an example of connecting a Macintosh file to the C++ stream library, so that I can operate on either fork with the iostream library?

By default the Apple-ported iostream libraries only work with the data fork. You should use the Resource Manager traps for resource handling, for two reasons: (1) they are tested and work well, and (2) any outside file I/O packages can't by default handle two forks (because UNIX and MS-DOS only know of one single data fork in a file).

However, the AT&T streams package has something called `strstreams`, which you can use to redirect byte streams from one memory location to another. If you want to get a form of resource fork handling with streams support, you could use the strstreams concept and redirect material sent to a memory location to/from the resource fork. It needs a little bit of coding, but the MPW C++ 3.1 documentation about streams provides the starting information.

C++ static constructors and checking for an FPU

Date Written: 8/17/92

Last reviewed: 6/14/93

We are faced with a problem related to the sequencing of events when our application starts up: Because of the way that static constructors are handled in C++, code gets executed before our main is encountered, thus we're unable to check for an FPU before the code that uses the FPU is executed, which will of course crash on an FPUless machine (a Bad Thing). We'd like to insert a call to code that can check for the presence of the FPU and take appropriate steps before `__CplusInit` is called from `_RTInit`. Is there a way to do this? One possibility involves messing with the library to rename `__CplusInit` to something else and writing our own `__CplusInit` that will call the real one if the FPU checks are passed. Is there a better way to do this?

You're right, checking for an FPU can be a problem because of the C++ static constructors. The best workaround we can recommend to you at this time is the one you suggest of having your own `CplusInit`, which then calls the real one only after FPU checks are passed.

Back to top

Downloadables

Acrobat version of this Note (K). Download

Technical Notes by Date | Number | Technology | Title
Developer Documentation | Technical Q&As | Development Kits | Sample Code