POWERPLANT WORKSHOP

By Aaron Montgomery

Debugging Basics

How one goes about writing a PowerPlant application

About the author...

Aaron teaches in the Mathematics Department at Central Washington University in Ellensburg, WA. Outside of his job, he spends time riding his mountain bike, watching movies and entertaining his wife and two sons. You can email him at `montgoaa@cwu.edu`, try to catch his attention in the newsgroup `comp.sys.mac.oop.powerplant` or visit his web site at `mac69108.math.cwu.edu:8080/`.

THESE ARTICLES

This is the second article in a series of articles about Metrowerks' PowerPlant application framework. The first article provided an introduction to how the framework deals with commands. This article focuses on the debugging classes in the framework. This series assumes familiarity with the C++ language, the Macintosh Toolbox API and the CodeWarrior IDE. The articles were written using CodeWarrior 6 with the net-update to IDE 4.1.0.3 and no other modifications.

WHY DEBUGGING NOW?

There are (at least) three reasons not to do debugging so early in this series of articles. First, debugging doesn't directly solve the problem of writing the code for the application. Second, the debugging classes use a lot of advanced PowerPlant concepts. Third, the debugging classes are in the `_In Progress` folder and they may change dramatically between CodeWarrior releases. However, there are (at least) two better reasons to talk about debugging now instead of waiting. First, debugging code should be written as the program is written instead of retrofitting it to existing code. Second, the debugging tools can also be used to learn about how the framework operates. In particular, the debugging classes in PowerPlant can be used to examine the command chain (discussed in the first article) and the visual hierarchy (which will be discussed in third article).

ARTICLE LAYOUT

Instead of following the code straight through from start to finish, this article is organized by topic. We will quickly describe changes to the project and the `main()` function. Most of this can be transplanted from one project to the next without modification. Then we will cover the various facets of the Debugging Classes. It is probably best to read this article at least twice since many times topics will appear in the text prior to the point where they are explained.

CHANGES TO THE PROJECT

Although the project for the first article had both a Debug and Final build, there was little to distinguish them from each other. When setting up the debug projects you will need to include the Debugging Classes (which are included in the Advanced stationary option). If you are attempting to add these classes to an existing project, you need to b a little careful. The files `UDebugging.cp` and `UDebuggingPlus.cp` implement the same functions. You should include `UDebugging.cp` in the Final builds and `UDebuggingPlus.cp` in the Debug builds. The resource files `PP Debug Alerts.rsrc` and `PP Debug Support.rsrc` also duplicate resources, the first should be used in the Final builds and the second in the Debug builds. I also did the same for the `LDebugStream` class: the header `LDebugStream.h` has two implementations: `LDebugStream.cp` for Debug builds and `LFinalStream.cp` for Final builds.

Included with the Advanced stationary is an "Utilities" folder with some source code. I have renamed the folder to "Hsoi's Utilities" since he wrote these sources and I haven't changed them much. These implement new and delete using some of the PowerPlant features. I also added the access path to `HackedPlant`, where I've placed some modified PowerPlant source files. Comments about what changed and why are included in these files. If you are going to use these changes in all your projects, it might be worthwhile using a Source Tree for the `HackedPlant` files so that multiple projects can use them.

Once these changes have been made, the difference between these two targets is the setting of two macros and a lot of conditional compilation. The macro `NDEBUG` should only be defined for the Final target and the macro `PP_Debug` should be set to 1 for the Debug version and to 0 for the Final version. In order to avoid conditional compilation within the main source files, most of the headers provide macros that depend on the `PP_Debug` setting. The PowerPlant framework uses macros to avoid cluttering your main code with conditional compilation and I have added the `CDebuggingUtilities` class in order to remove even more conditionals from the main files. As a warning, I don't intend to describe how the `CDebuggingUtilities` methods do what I claim they do. The file is documented with commentary if you are curious.

Four support packages can be used with PowerPlant's debugging classes: MoreFiles, QC, Spotlight and DebugNew. The standard stationary and this article use only DebugNew and none of the others. If you have these (MoreFiles <`http://members.aol.com/jumplong/`> is free, QC and Spotlight <`http://www.onyx-tech.com/`> are commercial), you can include their source files and libraries and enable support for them in the prefix files.

ADDING SOME BUGS

The first thing I did to the project was to add some bugs. This involves creating a menu and adding the code to handle the bugs I introduced. The menu can be created in Constructor (the resource editor provided for PowerPlant). The steps involved with Constructor are not that different from any either ResEdit or Resourcerer. To add a menu, you would open the `AppResources.ppob` file, select the Menus (MENU) item in the window and then created a new menu resource (**New Menu Resource** in the **Edit** menu). Unless you want the menu to be titled `untitled`, you would select the new menu and change its name to **Bad Things** and its ID number to 1000. You could do this directly in the `AppResources.ppob` window or you could open an **Inspector** window from the **Window** menu. In general, if you select resource IDs above 999 you won't conflict with any of the PowerPlant resources. Now double-click the menu. This will open the Menu editing window. You would then add new menu items (using **New Menu Item** in the **Edit** menu). Next, change the menu names and add command numbers using the tab key (or an Inspector window).

Next you would modify the `FindCommandStatus()` and `ObeyCommand()` code to indicate that these should be enabled and to implement the bugs. This was done in the `CDocumentApp` class and you can view the code to see how extra cases were added to the switch statements in these methods. We will discuss exactly what the code inside each branch does later in this article, suffice it to say these are things you typically would not want your application to do.

CODE CHANGES MAIN.CP

The code changes in the `main()` function are not specific to this particular PowerPlant application. As a result the code in this project can serve as a template for any other applications you write. Due to space considerations, I'll only do a quick overview of what the code does here, leaving some topics for later in the article and others unexplored completely. In the source file, the comments from last month's article is indicated by comments beginning with `//1` while the comments which have been introduced for this article is indicated by `//•`.

main() in main.cp

```
int main()
{
  (void) set_new_handler(PP_NewHandler);

  try
  {
    CDebugUtilities::SetAction();

    SLResetLeaks_();
    DebugNewForgetLeaks_();

    InitializeHeap(5);
    UQDGlobals::InitializeToolbox();
    ::FlushEvents(everyEvent, nil);
    UEnvironment::InitEnvironment();

    CDebugUtilities::CheckEnvironment();

    LGrowZone* theGrowZone = NEW LGrowZone(20000);
    ValidateObject_(theGrowZone);
    SignalIf_(theGrowZone->MemoryIsLow());

    {
      CDocumentApp theApp;
      theApp.Run();
    }

    CDebugUtilities::PowerPlantCleanUp();
    DebugNewReportLeaks_();
    CDebugUtilities::ShowLeaks();
  }
  catch (...)
  {
    SignalStringLiteral_("Exception caught in main");
  }

  return 0;
}
```

The first thing we do is to install our own handler for `new`. This handler is designed to use other classes in the PowerPlant framework to handle low memory situations. All the `CDebugUtilities` methods will be inlined no-ops in the Final build, we discuss what they will do in the Debug build. `SetAction()` will establish the behavior of `Throw_` and `Signal_` macros (discussed in a later section). `SLResetLeaks_()` and `DebugNewForgetLeaks_()` are used in debugging memory (discussed in a later section). After this we call the standard initialization routines for a PowerPlant application, adding a call to `CDebugUtilities`' method `CheckEnvironment()` which confirms that we are running on a system supporting our Debugging Classes.

The `LGrowZone` class is designed to provide an extra memory cushion when the system has run out of ways to provide us with memory. You should read the comments at the top of the `LGrowZone.cp` file to take full advantage of the `LGrowZone` object but for this project we'll just set aside 20,000 bytes of space. The `ValidateObject_` is a memory debugging macro and the call to `Signal_` will raise a `Signal_` if we could not create the cushion for some reason.

The placement of the `CDocumentApp` inside its own scope insures that it will be destructed prior to checking for memory leaks. A number of PowerPlant objects are usually not destructed until application termination, but they will show up as leaks while you are debugging. The `PowerPlantCleanUp()` method will delete all of these objects so that you do not have any phantom leaks showing up in your log files. Finally `DebugNewReportLeaks_()` will write any leaks found to a log file and `ShowLeaks()` will open that log file in CodeWarrior if any leaks were found (the code for `ShowLeaks()` was posted in the PowerPlant newsgroup by David Phillip Oster).

CDocumentApp Changes

There are a few stock changes to the `CDocumentApp` classes that need to be made. Again, like the changes in the function `main()`, I won't describe how they do what I say they do but you can read the documentation in the source file's commentary.

CDocumentApp() in CDocumentApp.cp

```
CDocumentApp::CDocumentApp()
{
  if (UEnvironment::HasFeature(env_HasAppearance)) {
    ::RegisterAppearanceClient();
  }

  CDebugUtilities::AddSIOUXAttachment(this);

  UControlRegistry::RegisterClasses();
  CTextDocument::RegisterClasses();

}
```

We will attach two attachments to the `CDocumentApp` object in order to aid our debugging. Attachments intercept the `ObeyCommand()` method and are able to extend the number of commands a `LCommander` is able to handle. This ability to extend any `LCommander`'s list of commands without changing the `LCommander`'s source code is a powerful technique for code reuse and you should read the more complete discussion of attachments in Chapter 15 of **The PowerPlant Book** or in the March 1999 *MacTech* article by John C. Daub.

The `LSIOUXAttachment` allows us to use a console window for the streams `cout` and `cerr`. Although I have found that it sometimes interferes with the program, you may find it useful and so I have given an example of its use here. This should only be used in the Debug builds and so I created a `CDebugUtilities` method to be called in the `CDocumentApp`'s constructor to handle the conditional compilation.

The other new code here is a call to `UControlRegistry::RegisterClasses()`. This call is needed to prepare for the dialogs that the debugging classes will use. We will discuss this need for registering classes in the next article. The other place an attachment is added is in the `Initialize()` method.

Initialize() in CDocumentApp.cp

```
void
CDocumentApp::Initialize()
{
  LDocApplication::Initialize();

  CDebugUtilities::AddDebugMenuAttachment(this);
}
```

The first thing to be done is to call `Initialize()` for the base class of `CDocumentApp`. This should always be your first call in your `Initialize()` methods. The `LDebugMenuAttachment` provides the menu interface to the debugging classes. Since it must be attached after the menu bar has been initialized, it needs to be added in the `CDocumentApp`'s `Initialize()` method (not the constructor).

THROW_ AND SIGNAL_

The most basic of the debugging tools available are `Throw_` and `Signal_`. The `Throw_` macros should be used when a problem occurs that must be handled (for example, if the application runs out of memory). The `Signal_` macros should be used when a problem occurs which (while unexpected) should not cause a problem in the final version (for example, if MacsBug is not installed). You can find a number of `Throw_` and `Signal_` macros in the header file `UException.h`.

The `Debug_Throw` macro determines the behavior of `Throw_` and the `Debug_Signal` macro determines the behavior of `Signal_`. If `Debug_Throw` is undefined, then `Throw_` will cause a C++ exception to be thrown. If `Debug_Signal` is undefined, then `Signal_` will do nothing. On the other hand, if the macros are defined, the behavior of `Throw_` and `Signal_` depend on a run-time variables of type

```
type enum {
  debugAction_Nothing,
  debugAction_Alert,
  debugAction_Debugger
} EDebugAction;
```

The current setting is established by calls to `SetDebugThrow_()` and `SetDebugSignal_()`. In the case of `debugAction_Nothing`, the behavior will be the same as that in the Final build. In the case of `debugAction_Alert`, an alert will be displayed. Once the alert is displayed you can choose one of the following options: Log, Quiet, Quit, Debugger, Continue. Choosing Log will log the exception to a log file. Choosing Quiet will change the setting to `debugAction_Nothing`. Choosing Debugger will drop into your debugger (but not change the setting). Choosing Quit will quit the application and choosing Continue will continue execution. In the case of `debugAction_Debugger`, you will be dropped into the debugger. You can learn a lot about how these work by using the Throw and Signal items in the **Bad Things** menu while changing the behavior with the **gDebugThrow** and **gDebugSignal** items in the Debugging menu (the one whose title is a bug icon). You should also experiment with the `StDisableSignal_()` and `StDisableThrow_()` objects which temporarily disable this behavior (examples are available in the **Bad Things** menu).

DEBUG NEW

DebugNew is a collection of routines to help you debug your application for memory errors (leaks, overwriting arrays, dangling pointers). The call to `DebugNewForgetLeaks_()` in `main()` tells DebugNew to forget all allocations made prior to that point in the application. The call to `DebugNewReportLeaks_()` will write out a log file indicating every memory allocation which has not been released up and `ShowLeaks()` will open this log file if it contains information about any leaks. If you play with the **Leak with NEW** and **Leak with new** menu items and then quit the application (while leaving CodeWarrior running), the leaks log should open. In it you can see that by calling `NEW` you will get the file name and line number where the leaked memory was allocated. The leaks from `new` are also recorded, but you get no information about where they occur.

The macros `DELETE` and `DELETE_ARRAY` can be used in place of `delete` and `delete[]` and they will delete the pointer and then set it to `nil`. All the error handling has been built into the `delete` operators so you can use them if you prefer. The two `DELETE` macros replace the `DisposeOf_` macros provided by PowerPlant for two reasons: first, most of the error checking was redundant when using `PP_DebugNew.cp` and second, the non-redundant error checking was incorrectly written.

You should play with the various memory related menu items in this section of the **Bad Things** menu to get a feel for the type of information DebugNew can provide. One thing I have noticed is that if the pointer is determined to be invalid by DebugNew, its memory will not be reclaimed when you call `delete`. This means that overwritten arrays or mismatched new/delete calls will add lines to your leaks log. Fixing the overwrite or mismatch problem will fix the leak.

LDebugStream

The `LDebugStream` class is designed for streaming debug information to a variety of locations. The most common location would be a log file, however, you can also stream to a `Throw_`, `Signal_` or directly to the debugger. If you enable streaming to a console (as I did in this application), you will be able to stream to a console window of your application. The original implementation required conditional compilation in your source code between your Debug build and your Final build. I have removed the need for that by adding no-op implementations in `LFinalStream.cp` (which is compiled and linked in the Final builds).

The **Bad Things** menu has a number of items that stream to various locations. You will need to be careful when streaming into a `Throw_`, `Signal_` or to the debugger since these have a limited amount of space to display the information. Streaming to a console has some limitations also. If the console is on the screen, all command-key combinations are sent to the console first (before the `CDocumentApp` can act on them). In the case of many commands, the console will ignore them and the commands will get lost. You can see this by first streaming to the console and then typing **command-N**. A new window will not open as long as the console window is the front window.

Given the above limitations, you might guess that most of your streaming will be done to a file and you are probably right. All of the streaming from within the PowerPlant classes is done to a log file that will be placed in the same directory as your application.

Other Debugging Menu Options

That finishes the **Bad Things** menu, but there are a few items in the Debugging menu which deserve mention. The **Command Chain** window provides you with a visual representation of the commanders in the application (these were discussed in the previous article). It would be nice if it also listed all the attachments since they affect the command handling but I'll leave that modification as an exercise for the reader. We will discuss the **Visual Hierarchy** menu item next week, but if you play around with it, you can probably see figure out what it does. There are a number of Heap Routines for Compacting and Purging the Heap (on command or at regular intervals). These can be useful for stress testing your application. It would be nice to have a similar repeater for DebugNew Validate All, but again, I'll leave this as an exercise for the reader.

Finally, you can use the **Eat Memory…** item to determine how your application will handle in low memory situations. One way to do this is to use the **Launch Zone Ranger** item, check the number of free bytes in the application, round down to the nearest 1000 and then eat that much memory. This should cause the application to put up a low memory warning.

Concluding Remarks

We're slowly putting together an application that will be able to do something. Although this month's article seems somewhat off the shortest path, the tools in the Debugging menu will help with some of the topics planned for the third article (windows). A natural question that arises now (if not earlier) is: How much longer until I can really do something? My prediction is after the next three articles (windows, files, dialogs) you should be well on your way. If you can't wait that long, try reading some of **The PowerPlant Book**.

POWERPLANT REFERENCES

References that are particularly appropriate for this article are the following:

"Applications and Events" chapter in **The PowerPlant Book** (it has a section on debugging)

"Debugging in PowerPlant" chapter in **PowerPlant Advanced Topics**

John C. Daub's "Modifying Objects at Runtime in PowerPlant" in *MacTech*, March 1999

John C. Daub's "PowerPlant's Debugging Classes" in *MacTech*, May 1999

`PP_DebugMacros.h`, `UDebugNew.h`, `UHeapUtils.h`, and `UOnyx.h` header files