

According to Hoyle...

Mac OS X Development Beyond Leopard

by Jonathan Hoyle

jhoyle@maccompanion.com

macCompanion

November 2007

"*Beyond* Leopard?!? But Leopard just came out!"

With Apple's highly anticipated release of Mac OS X 10.5 Leopard on October 26, there will no doubt be a plethora of reviews, insights, tips for you to enjoy, including many great ones here in macCompanion. Over the past few months, we have used this column to help prepare developers for the Leopard-skinned future. These include reviews of WWDC, Leopard, Xcode and Objective-C 2.0:

- <http://www.maccompanion.com/archives/July2007/Columns/AccordingtoHoyle23.htm>
- <http://www.maccompanion.com/archives/August2007/Columns/AccordingtoHoyle.htm>
- <http://www.maccompanion.com/macc/archives/September2007/Columns/AccordingtoHoyle.htm>
- <http://www.maccompanion.com/macc/archives/October2007/Columns/AccordingtoHoyle.htm>

Now that Leopard is in the present, it is sometimes easy to forget that there is still a future ahead of us. There is always something just around the corner, and this is one thing software developers like to pride themselves for being prepared. Knowing what is coming next will often help you today, especially if you plan to keep your application selling after release 1.0. For this reason, we will take this month to get greedy and look just beyond the confines of this first release of Leopard. So if you will indulge me this momentary breach of etiquette, come as we take an unauthorized tour into the near future of Mac OS X software development.

Xcode & gcc

With the release of Mac OS X 10.5 Leopard, Apple has provided Mac software developers the latest upgrade of its tools: Xcode 3. Although there are significant improvements in the IDE, its main component - the gcc compiler - remains unchanged from Mac OS X 10.4 Tiger. As you may already be aware, gcc is an open source compiler, part of the Gnu project sponsored by the Free Software Foundation. Apple's Xcode is the integrated development environment which wraps this compiler. Like gcc, Xcode is free; unlike gcc, however, Xcode is neither cross-platform nor open source.

Xcode 2.x and 3 each use gcc version 4.0.1 as its default compiler, although Xcode does allow you to change to the older gcc 3.3 compiler. Given that gcc 3.3 is not fully ANSI C++ compliant and lacks Intel compatibility, there is little reason for the modern developer to consider gcc 3.3 today. And even when version 3.3 was the default compiler in Xcode 1.5, Apple was recommending its developers to compile and test against gcc 4.0, so as to be prepared for the future. After having taken great pains to migrate Mac programmers to gcc 4, Apple decided not to make any compiler changes with Xcode 3 and Leopard, at least not just yet.

Despite Apple's hesitation to move beyond version 4.0.1, Gnu has been moving forward quite readily on its own. Version 4.0.x updates continued in 2005, with 4.1 released in February 2006, and a succession of 4.1.x updates following thereafter. 4.2 was released in March of this year, with 4.2.1 in July, and 4.2.2 in October. How long will Apple stay with version 4.0.1 before it decides to upgrade? This is of course a matter of speculation, but it seems likely that they will be making a move sooner rather than later. Those Mac developers who have signed up for their free ADC membership can see for themselves Apple's early progress on gcc 4.2.

Apple's **GCC 4.2 Developer Preview** is still under non-disclosure, so I cannot speak on its particulars. (Irrespectively, a shrewd Google search will yield a lot of information for you, including the *GCC 4.2 Developer Preview Release Notes* themselves.) You'll note that the NDA on Apple's Developer Preview is not much of a limitation, since gcc 4.2 is itself freely available and all information on it is open source. Essentially, I can speak on Xcode 3, and I can speak on gcc 4.2. I just can't speak of gcc 4.2 within Xcode 3. Therefore, I will address these separately.

Xcode 3: Where We Are Today?

Xcode 3 has added a number of powerful new features that improve the Mac development experience. The most notable is the change of the default debugging format from STABS to DWARF. In STABS, debugging information is hard-linked into the actual executable itself, grossly inflating file size. The STABS portion was typically much larger than the executable itself and thereby slowing down the link process. DWARF only soft-links this information, making the executable much smaller. DWARF has the additional benefit of being more C++ friendly, handling namespaces and inline functions much better. For those still using Xcode 2.3 or higher, DWARF is available (although off by default); my recommendation is for all users of Xcode 2.x to switch to DWARF now.

Another significant improvement in Xcode 3 is its increased security. One such security improvement is the inclusion of what's called "stack canaries". A stack canary is essentially a guard band between scope variables and a function's return addresses. Object size checking is also done, by replacing (for example) strcpy() with a parameter checking equivalent. Neither of these new features is turned on by default, presumably to allow for better code optimization. You must turn these items on in your Project settings.

Perhaps the best guard against malicious programs is in the treatment of space allocated by `malloc()`. In previous versions of Xcode, such space could be used to hold executable code. This is rarely desired by the developer, and is often exploited by malware copying unpleasant code into its contents. With Xcode 3, `malloc()`-ed space is non-executable by default. Developers who wish to make it executable memory must use `mprotect()`.

gcc 4.2: Where We Will Be Tomorrow

When looking at the change list of gcc over the past two years, we see that there are a number of improvements made since the version found in Xcode. As stated earlier, learning about these and other changes in gcc do not require you to sign an NDA with Apple, as Gnu publicly releases this information as gcc evolves. Gnu has separate web pages documenting the modification history for gcc 4.0.x (last version 4.0.4), 4.1.x (latest 4.1.2), and 4.2.x (latest 4.2.2 as I write this), and you can view them here:

gcc 4.0 Release History: <http://gcc.gnu.org/gcc-4.0/changes.html>

gcc 4.1 Release History: <http://gcc.gnu.org/gcc-4.1/changes.html>

gcc 4.2 Release History: <http://gcc.gnu.org/gcc-4.2/changes.html>

Rather than parrot what is already in these release notes, I wish to concentrate on two of the more powerful features in 4.2 that will likely make your life easier as a Mac developer. Both of these are changes specific with 4.2.x.

Parallel Universe

One of the most powerful new features of gcc 4.2 is *OpenMP*, a multiprocessor support capability. Consider a simple, yet common, task of summing the values of two arrays and placing their sum in a third array:

```
for (i = 0; i < numItems; i++)
    z[i] = x[i] + y[i];
```

Suppose this code snippet lives in a critical area of code which you would like to optimize. Machines with multiple processors could be used to parallelize this loop, by dividing the work across each processor core, dramatically improving performance. Unfortunately, writing the threading code and implementing the required mutexes to support this is quite complex, and would necessarily require a great deal of code debugging to get it right. If such a procedure is needed throughout different areas of the code, it gets even worse.

With OpenMP, however, only two directives need be added:

```
// parallelize this loop
#pragma omp parallel shared(x,y,z,chunk) private(i)
{
    #pragma omp for schedule(dynamic, chunk) nowait
    for (i = 0; i < numItems; i++)
        z[i] = x[i] + y[i];
}
```

The first `#pragma` informs the compiler which data objects are being shared across threads, and which is private. The second `#pragma` handles the `for` loop chunking. This code is an order of magnitude easier to write, as it simply directs the compiler to perform the parallelization. *OpenMP* is implemented by POSIX threads, thus making them quite solid and secure.

Visible Means of Support

In today's software development arena, programmers find that they must increasingly interface with other programs. Frequently, this means loading shared libraries written by other people. Unfortunately, not all of these programs are well modularized, and global functions and variables used in these libraries may have generic names which can easily clash. Such short-sighted authors declare global variables, functions and types with names like **Max**, **Run**, or even something as embarrassing as **i**. Even careful programmers may accidentally hit upon name collisions unintentionally. Authors of shared libraries must struggle between using obscure names unlikely to clash and keeping their source code readable.

The C++ mechanism for avoiding such name collisions is to encapsulate related the objects into a namespace. Items within one namespace do not collide with identically named ones in another namespace. A minor risk that remains is that two libraries might use the same namespace name, such as **MyNamespace** (although this is admittedly pretty unlikely). The bigger problem is simply that many developers are either ignorant of, or simply too lazy to use namespaces.

In gcc 4.0, this problem can be handled with the visibility attributes. A library author can define his structure with the `__attribute__((visibility("hidden")))` suffix, as so:

```
    // hidden structure
    struct InternalStruct
__attribute__((visibility("hidden")))
    {
        int    field1;
        double field2;
        void   *field3;
    }
```

When doing so, no one outside this library will see this structure defined, The gcc 4.2 compiler takes this functionality to the next level by automatically hiding functions and variables associated with a hidden type:

```
    void    MyFunction(InternalStruct    &inStruct);    //
automatically hidden

    std::vector<int>                myVector1;    // not
hidden

    std::vector<InternalStruct>    myVector2;    //
automatically hidden
```

C++ developers may even hide entire namespaces in the same manner:

```
    namespace hiddenSpace
__attribute__((visibility("hidden")))
    {
        // implementation details all hidden
    }
```

Anonymous namespaces have no name, so it is not possible to associate it with this visibility attribute. To address this, gcc 4.2 implicitly makes all items in an anonymous namespace (as well as items outside it that use them) static:

```
// anonymous namespace
namespace
{
    struct myStruct
    {
        ...
    };
}

void foo(MyStruct &myParm); // static
std::vector<MyStruct> myVector; // static
```

These visibility improvements do more than simply avoid name collisions. It also significantly improves linker performance, as fewer variables and types need be considered whilst building your program.

Stepping into the Future

These are merely two of the more significant changes found in gcc 4.2. There are of course many others. If you wish to be prepared for the next generation compiler, my recommendation is to keep an eye on gcc 4.2. It is this author's opinion that it will not be long before the future will (once again) be here.

Coming Up: Next month, another exciting episode in the According to Hoyle. See you in 30!