
Code Size Performance Guidelines

[Performance](#) > [Tools](#)



2006-06-28



Apple Inc.
© 2003, 2006 Apple Computer, Inc.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, mechanical, electronic, photocopying, recording, or otherwise, without prior written permission of Apple Inc., with the following exceptions: Any person is hereby authorized to store documentation on a single computer for personal use only and to print copies of documentation for personal use provided that the documentation contains Apple's copyright notice.

The Apple logo is a trademark of Apple Inc.

Use of the "keyboard" Apple logo (Option-Shift-K) for commercial purposes without the prior written consent of Apple may constitute trademark infringement and unfair competition in violation of federal and state laws.

No licenses, express or implied, are granted with respect to any of the technology described in this document. Apple retains all intellectual property rights associated with the technology described in this document. This document is intended to assist application developers to develop applications only for Apple-labeled computers.

Every effort has been made to ensure that the information in this document is accurate. Apple is not responsible for typographical errors.

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Apple, the Apple logo, Carbon, Cocoa, Mac, Mac OS, Pages, and Xcode are trademarks of Apple Inc., registered in the United States and other countries.

Simultaneously published in the United States and Canada.

Even though Apple has reviewed this document, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY

DEFECT OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Contents

Introduction to Code Size Performance Guidelines 7

Organization of This Document 7

Overview of the Mach-O Executable Format 9

The __TEXT Segment: Read Only 9
The __DATA Segment: Read/Write 10
Mach-O Performance Implications 10

Managing Code Size 13

Compiler-Level Optimizations 13
Additional Optimizations 14
 Dead Strip Your Code 14
 Strip Symbol Information 15
 Eliminate C++ Exception Handling Overhead 15
 Avoid Excessive Function Inlining 16
 Build Fixed-Position Application Code 17
 Build Frameworks as a Single Module 17

Improving Locality of Reference 19

Profiling Code With gprof 19
 Generating Profiling Data 19
 Generating Order Files 20
 Fixing Up Your Order Files 21
 Linking with an Order File 21
 Limitations of gprof Order Files 22
Profiling With the Monitor Functions 22
Organizing Code at Compile Time 23
Reordering the __text Section 23
 Reordering Procedures 24
 Procedure Reordering for Large Programs 25
Reordering Other Sections 28
 Reordering Literal Sections 28
 Reordering Data Sections 29
Reordering Assembly Language Code 30

Reducing Shared Memory Pages 31

Declaring Data as const 31

Initializing Static Data 33
Avoiding Tentative-Definition Symbols 33
Analyzing Mach-O Executables 34

Minimizing Your Exported Symbols 35

Identifying Exported Symbols 35
Limiting Your Exported Symbols 36
Limiting Exports Using GCC 4.0 36

Document Revision History 37

Index 39

Tables and Listings

Overview of the Mach-O Executable Format 9

Table 1	Major sections in the __TEXT segment	9
Table 2	Major sections of the __DATA segment	10

Managing Code Size 13

Table 1	GCC compiler optimization options	13
---------	-----------------------------------	----

Improving Locality of Reference 19

Listing 1	Using monitor functions	23
Listing 2	Code for Unique.c	26

Introduction to Code Size Performance Guidelines

In the context of performance, there is a distinct correlation between memory usage and efficiency. The more memory your application occupies, the more inefficient it is going to be. More memory means more memory allocations, more code, and a greater potential for paging.

The focus of this programming topic is on the reduction of your executable code. Reducing your code footprint is not just a matter of turning on code optimizations in your compiler, although that does help. You can also reduce your code footprint by organizing your code so that only the minimum set of required functions is in memory at any given time. You implement this optimization by profiling your code.

Reducing the amount of memory allocated by your application is also important in reducing your memory footprint; however, that information is covered in *Memory Usage Performance Guidelines* in Performance Documentation.

Organization of This Document

This programming topic contains the following articles:

- [“Overview of the Mach-O Executable Format”](#) (page 9) describes how to use the organization of the Mach-O executable format to improve the efficiency of your code.
- [“Managing Code Size”](#) (page 13) describes several compiler options that you can use to reduce the overall size of your executables.
- [“Improving Locality of Reference”](#) (page 19) describes how to profile and reorganize your code to improve loading times for code segments.
- [“Reducing Shared Memory Pages”](#) (page 31) describes ways to reduce the size of your `__DATA` segments.
- [“Minimizing Your Exported Symbols”](#) (page 35) shows how you identify and eliminate unnecessary symbol information in your code.

Overview of the Mach-O Executable Format

Mach-O is the native executable format of binaries in Mac OS X and is the preferred format for shipping code. An executable format determines the order in which the code and data in a binary file are read into memory. The ordering of code and data has implications for memory usage and paging activity and thus directly affects the performance of your program.

A Mach-O binary is organized into segments. Each segment contains one or more sections. Code or data of different types goes into each section. Segments always start on a page boundary, but sections are not necessarily page-aligned. The size of a segment is measured by the number of bytes in all the sections it contains and rounded up to the next virtual memory page boundary. Thus, a segment is always a multiple of 4096 bytes, or 4 kilobytes, with 4096 bytes being the minimum size.

The segments and sections of a Mach-O executable are named according to their intended use. The convention for segment names is to use all-uppercase letters preceded by double underscores (for example, `__TEXT`); the convention for section names is to use all-lowercase letters preceded by double underscores (for example, `__text`).

There are several possible segments within a Mach-O executable, but only two of them are of interest in relation to performance: the `__TEXT` segment and the `__DATA` segment.

The `__TEXT` Segment: Read Only

The `__TEXT` segment is a read-only area containing executable code and constant data. By convention, the compiler tools create every executable file with at least one read-only `__TEXT` segment. Because the segment is read-only, the kernel can map the `__TEXT` segment directly from the executable into memory just once. When the segment is mapped into memory, it can be shared among all processes interested in its contents. (This is primarily the case with frameworks and other shared libraries.) The read-only attribute also means that the pages that make up the `__TEXT` segment never have to be saved to backing store. If the kernel needs to free up physical memory, it can discard one or more `__TEXT` pages and re-read them from disk when they are needed.

[Table 1](#) (page 9) lists some of the more important sections that can appear in the `__TEXT` segment. For a complete list of segments, see *Mach-O Runtime Architecture*.

Table 1 Major sections in the `__TEXT` segment

Section	Description
<code>__text</code>	The compiled machine code for the executable
<code>__const</code>	The general constant data for the executable
<code>__cstring</code>	Literal string constants (quoted strings in source code)
<code>__picsymbol_stub</code>	Position-independent code stub routines used by the dynamic linker (<code>dyld</code>).

The `__DATA` Segment: Read/Write

The `__DATA` segment contains the non-constant data for an executable. This segment is both readable and writable. Because it is writable, the `__DATA` segment of a framework or other shared library is logically copied for each process linking with the library. When memory pages are readable and writable, the kernel marks them *copy-on-write*. This technique defers copying the page until one of the processes sharing that page attempts to write to it. When that happens, the kernel creates a private copy of the page for that process.

The `__DATA` segment has a number of sections, some of which are used only by the dynamic linker. [Table 2](#) (page 10) lists some of the more important sections that can appear in the `__DATA` segment. For a complete list of segments, see *Mach-O Runtime Architecture*.

Table 2 Major sections of the `__DATA` segment

Section	Description
<code>__data</code>	Initialized global variables (for example <code>int a = 1;</code> or <code>static int a = 1;</code>).
<code>__const</code>	Constant data needing relocation (for example, <code>char * const p = "foo";</code>).
<code>__bss</code>	Uninitialized static variables (for example, <code>static int a;</code>).
<code>__common</code>	Uninitialized external globals (for example, <code>int a;</code> outside function blocks).
<code>__dylld</code>	A placeholder section, used by the dynamic linker.
<code>__la_symbol_ptr</code>	"Lazy" symbol pointers. Symbol pointers for each undefined function called by the executable.
<code>__nl_symbol_ptr</code>	"Non lazy" symbol pointers. Symbol pointers for each undefined data symbol referenced by the executable.

Mach-O Performance Implications

The composition of the `__TEXT` and `__DATA` segments of a Mach-O executable file has a direct bearing on performance. The techniques and goals for optimizing these segments are different. However, they have as a common goal: greater efficiency in the use of memory.

Most of a typical Mach-O file consists of executable code, which occupies the `__TEXT`, `__text` section. As noted in ["The `__TEXT` Segment: Read Only"](#) (page 9), the `__TEXT` segment is read-only and is mapped directly to the executable file. Thus, if the kernel needs to reclaim the physical memory occupied by some `__text` pages, it does not have to save the pages to backing store and page them in later. It only needs to free up the memory and, when the code is later referenced, read it back in from disk. Although this is cheaper than swapping—because it involves one disk access instead of two—it can still be expensive, especially if many pages have to be recreated from disk.

One way to improve this situation is through improving your code's locality of reference through procedure reordering, as described in ["Improving Locality of Reference"](#) (page 19). This technique groups methods and functions together based on the order in which they are executed, how often they are called, and the frequency with which they call one another. If pages in the `__text` section group functions logically in this way, it is

less likely they have to be freed and read back in multiple times. For example, if you put all of your launch-time initialization functions on one or two pages, the pages do not have to be recreated after those initializations have occurred.

Unlike the `__TEXT` segment, the `__DATA` segment can be written to and thus the pages in the `__DATA` segment are not shareable. The non-constant global variables in frameworks can have an impact on performance because each process that links with the framework gets its own copy of these variables. The main solution to this problem is to move as many of the non-constant global variables as possible to the `__TEXT,__const` section by declaring them `const`. [“Reducing Shared Memory Pages”](#) (page 31) describes this and related techniques. This is not usually a problem for applications because the `__DATA` section in an application is not shared with other applications.

The compiler stores different types of nonconstant global data in different sections of the `__DATA` segment. These types of data are uninitialized static data and symbols consistent with the ANSI C notion of “tentative definition” that aren’t declared `extern`. Uninitialized static data is in the `__bss` section of the `__DATA` segment. Tentative-definition symbols are in the `__common` section of the `__DATA` segment.

The ANSI C and C++ standards specify that the system must set uninitialized static variables to zero. (Other types of uninitialized data are left uninitialized.) Because uninitialized static variables and tentative-definition symbols are stored in separate sections, the system needs to treat them differently. But when variables are in different sections, they are more likely to end up on different memory pages and thus can be swapped in and out separately, making your code run slower. The solution to these problems, as described in [“Reducing Shared Memory Pages”](#) (page 31), is to consolidate the non-constant global data in one section of the `__DATA` segment.

Managing Code Size

The GCC compiler supports a variety of options for optimizing your code. Most of these techniques result in the generation of less code or faster code, depending on your needs. As you prepare your software for release, you should experiment with these techniques to see which ones benefit your code the most.

Compiler-Level Optimizations

As your project code stabilizes, you should begin experimenting with the basic GCC options for optimizing code. The GCC compiler supports optimization options that let you choose whether you prefer a smaller binary size, faster code, or faster build times.

For new projects, Xcode automatically disables optimizations for the development build style and selects the “fastest, smallest” option for the deployment build style. Code optimizations of any kind result in slower build times because of the extra work involved in the optimization process. If your code is changing, as it does during the development cycle, you do not want optimizations enabled. As you near the end of your development cycle, though, the deployment build style can give you an indication of the size of your finished product.

Table 1 lists the optimization levels available in Xcode. When you select one of these options, Xcode passes the appropriate flags to the GCC compiler for the given group or files. These options are available at the target-level or as part of a build style. See the Xcode Help for information on working with build settings for your project.

Table 1 GCC compiler optimization options

Xcode Setting	Description
None	The compiler does not attempt to optimize code. Use this option during development when you are focused on solving logic errors and need a fast compile time. Do not use this option for shipping your executable.
Fast	The compiler performs simple optimizations to boost code performance while minimizing the impact to compile time. This option also uses more memory during compilation.
Faster	Performs nearly all supported optimizations that do not require a space-time trade-off. The compiler does not perform loop unrolling or function inlining with this option. This option increases both compilation time and the performance of generated code.
Fastest	Performs all optimizations in an attempt to improve the speed of the generated code. This option can increase the size of generated code as the compiler performs aggressive inlining of functions. This option is generally not recommended. See “Avoid Excessive Function Inlining” (page 16) for more information.

Xcode Setting	Description
Fastest, smallest	Performs all optimizations that do not typically increase code size. This is the preferred option for shipping code as it gives your executable a smaller memory footprint.

As with any performance enhancement, do not make assumptions about which option will give you the best results. You should always measure the results of each optimization you try. For example, the “Fastest” option might generate extremely fast code for a particular module, but it usually does so at the expense of executable size. Any speed advantages you gain from the code generation are easily lost if the code needs to be paged in from disk at runtime.

Additional Optimizations

Besides code-level optimizations, there are some additional techniques you can use to organize your code at the module level. The following sections describes these techniques.

Dead Strip Your Code

For statically-linked executables, dead-code stripping is the process of removing unreferenced code from the executable file. The idea behind dead-stripping is that if the code is unreferenced, it must not be used and therefore is not needed in the executable file. Removing dead code reduces the size of your executable and can help reduce paging.

Starting with Xcode Tools version 1.5, the static linker (`ld`) supports dead stripping of executables. You can enable this feature directly from Xcode or by passing the appropriate command-line options to the static linker.

To enable dead-code stripping in Xcode, do the following:

1. Select your target.
2. Open the Inspector or Get Info window and select the Build tab.
3. In the Linking settings, enable the Dead Code Stripping option.
4. In the Code Generation settings, set the Level of Debug Symbols option to All Symbols.

To enable dead-code stripping from the command line, pass the `-dead_strip` option to `ld`. You should also pass the `-gfull` option to GCC to generate a complete set of debugging symbols for your code. The linker uses this extra debugging information to dead strip the executable.

Note: The “All Symbols” or `-gfull` option is recommended even if you do not plan to dead strip your code. Although the option generates larger intermediate files, it often results in smaller executables because the linker is able to remove duplicate symbol information more effectively.

If you do not want to remove any unused functions, you should at least isolate them in a separate section of your `__TEXT` segment. Moving unused functions to a common section improves the locality of reference of your code and reduces the likelihood of their being loaded into memory. For more information on how to group functions in a common section, see [“Improving Locality of Reference”](#) (page 19).

Strip Symbol Information

Debugging symbols and dynamic-binding information can take up a lot of space and comprise a large percentage of your executable’s size. Before shipping your code, you should strip out all unneeded symbols.

To strip debugging symbols from your executable, change the Xcode build-style settings to “Deployment” and rebuild your executable. You can also generate debugging symbols on a target-by-target basis if you prefer. See the Xcode Help for more information on build styles and target settings.

To remove dynamic-binding symbols manually from your executable, use the `strip` tool. This tool removes symbol information that would normally be used by the dynamic linker to bind external symbols at runtime. Removing the symbols for functions that you do not want to be dynamically bound reduces your executable size and reduces the number of symbols the dynamic linker must bind. Typically, you would use this command without any options to remove non-external symbols, as shown in the following example:

```
% cd ~/MyApp/MyApp.app/Contents/MacOS
% strip MyApp
```

This command is equivalent to running `strip` with the `-u` and `-r` options. It removes any symbols marked as `non-external` but does not remove symbols that are marked `external`.

An alternative to stripping out dynamic-binding symbols manually is to use an exports file to limit the symbols exported at build time. An exports file identifies the specific symbols available at runtime from your executable. For more information on creating an exports file, see [“Minimizing Your Exported Symbols”](#) (page 35).

Eliminate C++ Exception Handling Overhead

When an exception is thrown, the C++ runtime library must be able to unwind the stack back to the point of the first matching `catch` block. For this to work, the GCC compiler generates stack unwinding information for each function that may throw an exception. This unwinding information is stored in the executable file and describes the objects on the stack. This information makes it possible to call the destructors of those objects to clean them up when an exception is thrown.

Even if your code does not throw exceptions, the GCC compiler still generates stack unwinding information for C++ code by default. If you use exceptions extensively, this extra code can increase the size of your executable significantly.

Disabling Exceptions

You can disable exception handling in Xcode altogether by disabling the “Enable C++ Exceptions” build option for your target. From the command line, pass the `-fno-exceptions` option to the compiler. This option removes the stack unwinding information for your functions. However, you must still remove any `try`, `catch`, and `throw` statements from your code.

Selectively Disabling Exceptions

If your code uses exceptions in some places but not everywhere, you can explicitly identify methods that do not need unwinding information by adding an empty exception specification to the method declaration. For example, in the following code, the compiler must generate stack unwinding information for `my_function` on the grounds that `my_other_function` or a function called by it may throw an exception.

```
extern int my_other_function (int a, int b);
int my_function (int a, int b)
{
    return my_other_function (a, b);
}
```

However, if you know that `my_other_function` cannot throw exceptions, you can signal this to the compiler by including the empty exception specification (`throw ()`) in the function declarations. Thus, you would declare the preceding function as follows:

```
extern int foo (int a, int b) throw ();
int my_function (int a, int b) throw ()
{
    return foo (a, b);
}
```

Minimizing Exception Use

When writing your code, consider your use of exceptions carefully. Exceptions should be used to indicate exceptional circumstances—that is, they should be used to report problems that you did not anticipate. If you read from a file and got an end-of-file error, you would not want to throw an exception because this is a known type of error and can be handled easily. If you try to read from a file you know to be open and are told the file ID is invalid, then you would probably want to throw an exception.

Avoid Excessive Function Inlining

Although inline functions can improve speed in some situations, they can also degrade performance on Mac OS X if used excessively. Inline functions eliminate the overhead of calling a function but do so by replacing each function call with a copy of the code. If an inline function is called frequently, this extra code can add up quickly, bloating your executable and causing paging problems.

Used properly, inline functions can save time and have a minimal impact on your code footprint. Remember that the code for inline functions should generally be very short and called infrequently. If the time it takes to execute the code in a function is less than the time it takes to call the function, the function is a good candidate for inlining. Generally, this means that an inline function probably should have no more than a few lines of code. You should also make sure that the function is called from as few places as possible in your code. Even a short function can cause excessive bloat if it is made inline in dozens or hundreds of places.

Also, you should be aware that the “Fastest” optimization level of the GCC should generally be avoided. At this optimization level, the compiler aggressively tries to create inline functions, even for functions that are not marked as inline. Unfortunately, doing so can significantly increase the size of your executable and cause far worse performance problems due to paging.

Build Fixed-Position Application Code

By default, most code is built with the `-dynamic` compiler option. This option enables indirect symbol addressing and position-independent code generation, which allows the generated code to be relocated within the virtual memory space of the process. For projects such as bundles and frameworks, this option is required. The dynamic-linker must be able to relocate the bundle or framework and patch up symbol references at runtime.

Unlike bundles and frameworks, applications do not need the position-independent code generation feature provided by the `-dynamic` option. Application code is never relocated within the process space. However, it does still require the indirect addressing feature to allow for dynamic linking to other code modules, such as bundles. To solve this problem, the GCC versions 3.1 and later support the `-mdynamic-no-pic` option, which disables position-independent code generation but allows indirect symbol addressing. You should always enable this option when building applications.

Note: In Xcode, you specify the `-mdynamic-no-pic` option by choosing the Generate Position Independent Code option from the Code Generation settings.

Build Frameworks as a Single Module

Most shared libraries don't need the module features of the Mach-O runtime. In addition, cross-module calls incur the same overhead as cross-library calls. As a result, you should link all of your project's intermediate object files together into a single module.

To combine your object files, you must pass the `-r` option to `ld` during the link phase. If you are using Xcode to build your code, this is done for you by default.

Improving Locality of Reference

An important improvement you can make to your application's performance is to reduce the number of virtual memory pages used by the application at any given time. This set of pages is referred to as the **working set** and consists of application code and runtime data. Reducing the amount of in-memory data is a function of your algorithms, but reducing the amount of in-memory code can be achieved by a process called **scatter loading**. This technique is also referred to as improving the locality of reference of your code.

Normally, compiled code for methods and functions is organized by source file in the resulting binary. Scatter loading changes this organization and instead groups related methods and functions together, independent of the original locations of those methods and functions. This process allows the kernel to keep an active application's most frequently referenced executable pages in the smallest memory space possible. Not only does this reduce the footprint of the application, it reduces the likelihood of those pages being paged out.

Important: You should generally wait until very late in the development cycle to scatter load your application. Code tends to get moved around during development, which can invalidate prior profiling results.

Profiling Code With gprof

Given profiling data collected at runtime, `gprof` produces an execution profile of a program. The effect of called routines is incorporated in the profile of each caller. The profile data is taken from the call graph profile file (`gmon.out` by default), which is created by a program compiled and linked with the `-pg` option. The symbol table in the executable is correlated with the call graph profile file. If more than one profile file is specified, the `gprof` output shows the sum of the profile information in the given profile files.

The `gprof` tool is useful for many purposes, including the following:

- cases where the Sampler application doesn't work well, such as command-line tools or applications that quit after a short period of time
- cases where you want a call graph that includes all the code that might be called in a given program, rather than a periodic sampling of calls
- cases where you want to change the link order of your code to optimize the code locality

Generating Profiling Data

Before you can profile your application, you must set up your project to generate profiling information. To generate profiling information for your Xcode project, you must modify your target or build-style settings to include the "Generate profiling code" option. (See the Xcode Help for information on enabling target and build-style settings.)

The profiling code inside your program generates a file called `gmon.out` with the profiling information. (Usually, this file is placed in the current working directory.) To analyze the data in this file, copy it to the directory containing your executable prior to calling `gprof`, or just specify the path to `gmon.out` when you run `gprof`.

In addition to profiling your own code, you can find out how much time is spent in Carbon and Cocoa frameworks by linking against profile versions of those frameworks. To do this, add the `DYLD_IMAGE_SUFFIX` setting to your target or build style and set its value to `_profile`. The dynamic linker combines this suffix with the framework name to link against the profile version of the framework. To determine which frameworks support profiling, look at the frameworks themselves. For example, the Carbon library comes with both profile and debug versions.

Note: Profile and debug versions of libraries are installed as part of the developer tools package and may not be available on user systems. Make sure your shipping executable does not link against one of these libraries.

Generating Order Files

An order file contains an ordered sequence of lines, each line consisting of a source file name and a symbol name, separated by a colon with no other white space. Each line represents a block to be placed in a section of the executable. If you modify the file by hand, you must follow this format exactly so the linker can process the file. If the object file `name:symbol` name pair is not exactly the name seen by the linker, it tries its best to match up the names with the objects being linked.

The lines in an order file for procedure reordering consist of an object filename and procedure name (function, method, or other symbol). The sequence in which procedures are listed in the order file represents the sequence in which they are linked into the `__text` section of the executable.

To create an order file from the profiling data generated by using a program, run `gprof` using the `-S` option (see the man page for `gprof` (1)). For example,

```
gprof -S MyApp.profile/MyApp gmon.out
```

The `-S` option produces four mutually exclusive order files:

<code>gmon.order</code>	Ordering based on a “closest is best” analysis of the profiling call graph. Calls that call each other frequently are placed close together.
<code>callf.order</code>	Routines sorted by the number of calls made to each routine, largest numbers first.
<code>callo.order</code>	Routines sorted by the order in which they are called.
<code>time.order</code>	Routines sorted by the amount of CPU time spent, largest times first.

You should try using each of these files to see which provides the largest performance improvement, if any. See [“Using pagestuff to Examine Pages on Disk”](#) (page 25) for a discussion of how to measure the results of the ordering.

These order files contain only those procedures used during profiling. The linker keeps track of missing procedures and links them in their default order after those listed in the order file. Static names for library functions are generated in the order file only if the project directory contains a file generated by the linker's `-whatsloaded` option; see [“Creating a Default Order File”](#) (page 25) for details.

The `gprof -S` option doesn't work with executables that have already been linked using an order file.

Fixing Up Your Order Files

After you generate your order files, you should look through them and make sure they are correct. There are a number of cases where you need to edit your order files manually, including the following:

- Your executable contained assembly-language files.
- You profiled a stripped executable.
- Your executable contained files compiled without the `-g` option.
- Your project defines internal labels (typically for `goto` statements).
- You want to preserve the order of routines in a particular object file.

If the definition of a symbol is located in an assembly file, a stripped executable file, or a file compiled without the `-g` option, `gprof` omits the source file name from the symbol's entry in the order file. If your project uses such files, you must edit the order file manually and add the appropriate source filenames. Alternatively, you can delete the symbol references altogether to force the corresponding routines to be linked in default order.

If your code contains internal labels, you must remove those labels from the order files; otherwise, the function that defines the label will be split apart during the linking phase. You can prevent the inclusion of internal labels in assembly files altogether by prefixing them with the string `L_`. The assembler interprets symbols with this prefix as local to a particular function and strips them to prevent access by other tools such as `gprof`.

To preserve the order of routines in a particular object file, use the special symbol `.section_all`. For example, if the object file `foo.o` comes from assembly source and you want to link all of the routines without reordering them, delete any existing references to `foo.o` and insert the following line in the order file:

```
foo.o:.section_all
```

This option is useful for object files compiled from assembly source, or for which you don't have the source.

Linking with an Order File

Once you've generated an order file, you can link the program using the `-sectorder` and `-e start` options:

```
cc -o outputFile inputFile.o... -sectorder __TEXT __text orderFile -e start
```

To use an order file with a Xcode project, modify the “Other Linker Flags” option in the Deployment build style of your project. Add the text `-sectorder __TEXT __text orderFile` to this setting to specify your order file.

If any *inputFile* is a library rather than an object file, you may need to edit the order file before linking to replace all references to the object file with references to the appropriate library file. Again, the linker does its best to match names in the order file with the sources it is editing.

With these options, the linker constructs the executable file *outputFile* so that the contents of the `__TEXT` segment's `__text` section are constructed from blocks of the input files' `__text` sections. The linker arranges the routines in the input files in the order listed in *orderFile*.

As the linker processes the order file, it places the procedures whose object-file and symbol-name pairs aren't listed in the order file into the `__text` section of *outputFile*. It links these symbols in the default order. Object-file and symbol-name pairs listed more than once always generate a warning, and the first occurrence of the pair is used.

By default, the linker prints a summary of the number of symbol names in the linked objects that are not in the order file, the number of symbol names in the order file that are not in the linked objects, and the number of symbol names it tried to match that were ambiguous. To request a detailed listing of these symbols, use the `-sectorder_detail` option.

The linker's `-e start` option preserves the executable's entry point. The symbol `start` (note the lack of a leading `"_"`) is defined in the C runtime shared library `/usr/bin/crt1.o`; it represents the first text address in your program when you link normally. When you reorder your procedures, you have to use this option to fix the entry point. Another way to do this is to make the line `/usr/lib/crt1.o:start` or `/usr/lib/crt1.o:section_all` the first line of your order file.

Limitations of gprof Order Files

The `.order` files generated by `gprof` contain only those functions that were called or sampled when the executable was run. For library functions to appear correctly in the order file, a `whatloaded` file produced by the linker should exist in the working directory.

The `-S` option does not work with executables that have already been linked with an order file.

Production of the `gmon.order` file can take a long time—it can be suppressed with the `-x` parameter.

Filenames will be missing for the following items:

- files compiled without the `-g` parameter
- routines generated from assembly-language source
- executables that have had their debugging symbols removed (as with the `strip` tool)

Profiling With the Monitor Functions

The file `/usr/include/monitor.h` declares a set of functions that you can use to programmatically profile specific sections of your code. You can use these functions to gather statistics only for certain sections of your code, or for all of your code. You can then use the `gprof` tool to build call graph and other performance analysis data from the resulting file. [Listing 1](#) (page 23) shows how to use the monitor functions.

Listing 1 Using monitor functions


```
#include <monitor.h>

/* To start profiling: */
moninit();
moncontrol(1);

/* To stop, and dump to a file */
moncontrol(0);
monoutput("/tmp/myprofiledata.out");
monreset();
```

Organizing Code at Compile Time

The GCC compiler lets you specify attributes on any function or variable you declare. The `section` attribute lets you tell GCC which segment and section you want a particular piece of code to be placed.

 **Warning:** Do not use `section` attributes unless you understand the structure of Mach-O executables and know the rules for placing functions and data in the corresponding segments. Placing a function or global variable in an inappropriate section may break your program.

The `section` attribute takes several parameters that control where the resulting code is placed. At a minimum, you must specify a segment and section name for the code you want to place. Other options are also available and are described in the GCC documentation.

The following listing shows how you use the `section` attribute for a function. In this example, the `section` attribute is added to a forward declaration of the function. The attribute tells the compiler to place the function in a specific `__text` section of the executable.

```
void MyFunction (int a) __attribute__((section("__TEXT,__text.10")));
```

The following listing shows some examples of how you can organize your global variables using the `section` attribute.

```
extern const int x __attribute__((section("__TEXT,__my_const")));
const int x=2;

extern char foo_string[] __attribute__((section("__DATA,__my_data")));
char foo_string[] = "My text string\n";
```

For detailed information on specifying `section` attributes, see the documentation for the GCC compiler in `/Developer/Documentation/DeveloperTools/gcc3`.

Reordering the `__text` Section

As described in “[Overview of the Mach-O Executable Format](#)” (page 9), the `__TEXT` segment holds the actual code and read-only portions of your program. The compiler tools, by convention, place procedures from your Mach-O object files (with extension `.o`) in the `__text` section of the `__TEXT` segment.

As your program runs, pages from the `__text` section are loaded into memory on demand, as routines on these pages are used. Code is linked into the `__text` section in the order in which it appears in a given source file, and source files are linked in the order in which they are listed on the linker command line (or in the order specifiable in Xcode). Thus, code from the first object file is linked from start to finish, followed by code from the second file and third file and so on.

Loading code in the source file declaration order is rarely optimal. For example, suppose that certain methods or functions in your code are invoked repeatedly, while others are seldom used. Reordering the procedures to place frequently used code at the beginning of the `__text` section minimizes the average number of pages your application uses and thereby reduces paging activity.

As another example, suppose that all the objects defined by your code are initialized at the same time. Because the initialization routine for each class is defined in a separate source file, the initialization code is ordinarily distributed across the `__text` section. By contiguously reordering initialization code for all classes, you reduce the number of pages that need to be read in, enhancing initialization performance. The application requires just the small number of pages containing initialization code, rather than a larger number of pages, each containing a small piece of initialization code.

Reordering Procedures

Depending on the size and complexity of your application, you should pursue a strategy for ordering code that best improves the performance of your executable. Like most performance tuning, the more time you spend measuring and retuning your procedure order, the more memory you save. You can easily obtain a good first-cut ordering by running your application and sorting the routines by call frequency. The steps for this strategy are listed below and explained in more detail in the sections that follow:

1. Build a profile version of your application. This step generates an executable containing symbols used in the profiling and reordering procedures.
2. Run and use the application to create a set of profile data. Perform a series of functional tests, or have someone use the program for a test period.

Important: For best results, focus on the most typical usage pattern. Avoid using all the features of the application or the profile data might become diluted. For example, focus on launch time and the time to activate and deactivate your main window. Do not bring up ancillary windows.

3. Create order files. Order files list procedures in optimized order. The linker uses order files to reorder procedures in the executable.
4. Run the linker using the order files. This creates an executable with procedures linked into the `__text` section as specified in the order file.

For information on profiling your code and generating and linking an order file, see [“Profiling Code With gprof”](#) (page 19).

Procedure Reordering for Large Programs

For many programs, the ordering generated by the steps just described brings a substantial improvement over unordered procedures. For a simple application with few features, such an ordering represents most of the gains to be had by procedure reordering. However, larger applications and other large programs can benefit greatly from additional analysis. While order files based on call frequency or the call graph are a good start, you can use your knowledge of the structure of your application to further reduce the virtual-memory working set.

Creating a Default Order File

If you want to reorder an application's procedures using techniques other than those described above, you may want to skip the profiling steps and just start with a default order file that lists all the routines of your application. Once you have a list of the routines in suitable form, you can then rearrange the entries by hand or by using a sorting technique of your choice. You can then use the resulting order file with the linker's `-sectorder` option as described in ["Linking with an Order File"](#) (page 21).

To create a default order file, first run the linker with the `-whatloaded` option:

```
cc -o outputFile inputFile.o -whatloaded > loadedFile
```

This creates a file, *loadedFile*, that lists the object files loaded in the executable, including any in frameworks or other libraries. The `-whatloaded` option can also be used to make sure that order files generated by `gprof -S` include names for procedures in static libraries.

Using the file *loadedFile*, you can run `nm` with the `-onjls` options and the `__TEXT __text` argument:

```
nm -onjls __TEXT __text `cat loadedFile` > orderFile
```

The content of the file *orderFile* is the symbol table for the text section. Procedures are listed in the symbol table in their default link order. You can rearrange entries in this file to change the order in which you want procedures to be linked, then run the linker as described in ["Linking with an Order File"](#) (page 21).

Using `pagestuff` to Examine Pages on Disk

The `pagestuff` tool helps you measure the effectiveness of your procedure ordering by telling you which pages of the executable file are likely to be loaded in memory at a given time. This section briefly describes how to use this tool. See the `pagestuff` man page for more information.

The `pagestuff` tool prints out the symbols on a particular page of executable code. The following is the syntax for the command:

```
pagestuff filename [pageNumber | -a]
```

The output of `pagestuff` is a list of procedures contained in *filename* on page *pageNumber*. To view all the pages of the file, use the `-a` option in place of the page number. This output allows you to determine if each page associated with the file in memory is optimized. If it isn't, you can rearrange entries in the order file and link the executable again to maximize performance gains. For example, move two related procedures together so they are linked on the same page. Perfecting the ordering may require several cycles of linking and tuning.

Grouping Routines According to Usage

Why generate profile data for individual operations of your application? The strategy is based on the assumption that a large application has three general groups of routines:

- **Hot routines** run during the most common usage of the application. These are often primitive routines that provide a foundation for the application's features (for example, routines for accessing a document's data structures) or routines that implement the core features of an application, such as routines that implement typing in a word processor. These routines should be clustered together in the same set of pages.
- **Warm routines** implement specific features of the application. Warm routines are usually associated with particular features that user performs only occasionally (such as launching, printing, or importing graphics). Because these routines are used reasonably often, cluster them in the same small set of pages so they will load quickly. However, because there are long periods when users aren't accessing this functionality, these routines should not be located in the hot category.
- **Cold routines** are rarely used in the application. Cold routines implement obscure features or cover boundary or error cases. Group these routines together to avoid wasting space on a hot or warm page.

At any given time, you should expect most of the hot pages to be resident, and you should expect the warm pages to be resident for the features that the user is currently using. Only very rarely should a cold page be resident.

To achieve this ideal ordering, gather a number of profile data sets. First, gather the hot routines. As described above, compile the application for profiling, launch it, and use the program. Using `gprof -S`, generate a frequency sorted order file called `hot.order` from the profile data.

After creating a hot order file, create order files for features that users occasionally use, such as routines that only run when the application is launched. Printing, opening documents, importing images and using various non-document windows and tools are other examples of features that users use occasionally but not continually, and are good candidates for having their own order files. Naming these order files after the feature being profiled (for example, `feature.order`) is recommended.

Finally, to generate a list of all routines, build a "default" order file `default.order` (as described in ["Reordering Procedures"](#) (page 24)).

Once you have these order files, you can combine them using the code shown in [Listing 2](#) (page 26). You can use this listing to build a command-line utility that removes duplicate lines in the order files while retaining the ordering of the original data.

Listing 2 Code for Unique.c

```
//
// unique
//
// A command for combining files while removing
// duplicate lines of text. The order of other lines of text
// in the input files is preserved.
//
// Build using this command line:
//
// cc -ObjC -O -o unique -framework Foundation Unique.c
//
// Note that "unique" differs from the BSD command "uniq" in that
// "uniq" combines duplicate adjacent lines, while "unique" does not
```

```

// require duplicate lines to be adjacent. "unique" is also spelled
// correctly.
//

#import <stdio.h>
#import <string.h>
#import <Foundation/NSSet.h>
#import <Foundation/NSData.h>

#define kBufferSize 8*1024

void ProcessFile(FILE *fp)
{
    char buf[ kBufferSize ];

    static id theSet = nil;

    if( theSet == nil )
    {
        theSet = [[NSMutableSet alloc] init];
    }

    while( fgets(buf, kBufferSize, fp) )
    {
        id dataForString;

        dataForString = [[NSData alloc] initWithBytes:buf length:strlen(buf)];

        if( ![theSet containsObject:dataForString] )
        {
            [theSet addObject:dataForString];
            fputs(buf, stdout);
        }

        [dataForString release];
    }
}

int main( int argc, char *argv[] )
{
    int    i;
    FILE * theFile;
    int    status = 0;

    if( argc > 1 )
    {
        for( i = 1; i < argc; i++ )
        {
            if( theFile = fopen( argv[i], "r" ) )
            {
                ProcessFile( theFile );
                fclose( theFile );
            }
            else
            {
                fprintf( stderr, "Could not open '%s'\n", argv[i] );
                status = 1;
                break;
            }
        }
    }
}

```

```

        }
    }
}
else
{
    ProcessFile( stdin );
}
return status;
}

```

Once built, you would use the program to generate your final order file with syntax similar to the following:

```
unique hot.order feature1.order ... featureN.order default.order > final.order
```

Of course, the real test of the ordering is the amount by which paging I/O is reduced. Run your application, use different features, and examine how well your ordering file is performing under different conditions. You can use the `top` tool (among others) to measure paging performance.

Finding That One Last Hot Routine

After reordering you will usually have a region of pages with cold routines that you expect to be rarely used, often at the end of your text ordering. However, one or two hot routines might slip through the cracks and land in this cold section. This is a costly mistake, because using one of these hot routines now requires an entire page to be resident, a page that is otherwise filled with cold routines that are not likely to be used.

Check that the cold pages of your executable are not being paged in unexpectedly. Look for pages that are resident with high-page offsets in the cold region of your application's text segment. If there is an unwanted page, you need to find out what routine on that page is being called. One way to do this is to profile during the particular operation that is touching that page, and use the `grep` tool to search the profiler output for routines that reside on that page. Alternatively, a quick way to identify the location where a page is being touched is to run the application under the `gdb` debugger and use the Mach call `vm_protect` to disallow all access to that page:

```
(gdb) p vm_protect(task_self(), startpage_addr, vm_page_size, FALSE, 0);
```

After clearing the page protections, any access to that page causes a memory fault, which breaks the program in the debugger. At this point you can simply look at the function call stack (using the `bt` command) to learn why the routine was being called.

Reordering Other Sections

You can use the `-sectorder` option of the linker to organize blocks in most sections of the executable. Sections that might occasionally benefit from reordering are literal sections, such as the `__TEXT` segment's `__cstring` section, and the `__DATA` segment's `__data` section.

Reordering Literal Sections

The lines in the order file for literal sections can most easily be produced with the `ld` and `otool` tools. For literal sections, `otool` creates a specific type of order file for each type of literal section:

- For C string literal sections, the order-file format is one literal C string per line (with ANSI C escape sequences allowed in the C string). For example, a line might look like

```
Hello world\n
```

- For 4-byte literal sections, the order-file format is one 32-bit hex number with a leading 0x per line with the rest of the line treated as a comment. For example, a line might look like

```
0x3f8cccd (1.10000002384185790000e+00)
```

- For 8-byte literal sections, the order file line consists of two 32-bit hexadecimal numbers per line separated by white space each with a leading 0x, with the rest of the line treated as a comment. For example, a line might look like:

```
0x3ff00000 0x00000000 (1.00000000000000000000e+00)
```

- For literal pointer sections, the format of the lines in the order file represents the pointers, one per line. A literal pointer is represented by the segment name, the section name of the literal pointer, and then the literal itself. These are separated by colons with no extra white space. For example, a line might look like:

```
__OBJC:__selector_strs:new
```

- For all the literal sections, each line in the order file is simply entered into the literal section and appears in the output file in the order of the order file. No check is made to see if the literal is in the loaded objects.

To reorder a literal section, first create a “whatloaded” file using the `ld -whatloaded` option as described in section “[Creating a Default Order File](#)” (page 25). Then, run `otool` with the appropriate options, segment and section names, and filenames. The output of `otool` is a default order file for the specified section. For example, the following command line produces an order file listing the default load order for the `__TEXT` segment’s `__cstring` section in the file `cstring_order`:

```
otool -X -v -s __TEXT __cstring `cat whatloaded` > cstring_order
```

Once you’ve created the file `cstring_order`, you can edit the file and rearrange its entries to optimize locality of reference. For example, you can place literal strings used most frequently by your program (such as labels that appear in your user interface) at the beginning of the file. To produce the desired load order in the executable, use the following command:

```
cc -o hello hello.o -sectorder __TEXT __cstring cstring_order
```

Reordering Data Sections

There are currently no tools to measure code references to data symbols. However, you might know a program’s data-referencing patterns and might be able to get some savings by separating data for seldom-used features from other data. One way to approach `__data` section reordering is to sort the data by size so small data items end up on as few pages as possible. For example, if a larger data item is placed across two pages with two small items sharing each of these pages, the larger item must be paged in to access the smaller items. Reordering the data by size can minimize this sort of inefficiency. Because this data would normally need to be written to the virtual-memory backing store, this could be a major savings in some programs.

To reorder the `__data` section, first create an order file listing source files and symbols in the order in which you want them linked (order file entries are described at the beginning of [“Generating Order Files”](#) (page 20)). Then, link the program using the `-sectorder` command-line option:

```
cc -o outputFile inputFile.o... -sectorder __DATA __data orderFile -e start
```

To use an order file with a Xcode project, modify the “Other Linker Flags” option in the Deployment build style of your project. Add the text `-sectorder __DATA __data orderFile` to this setting to specify your order file.

Reordering Assembly Language Code

Some additional guidelines to keep in mind when reordering routines coded in assembly language:

- temporary labels in assembly code

Within hand-coded assembly code, be careful of branches to temporary labels that branch over a non temporary label. For example, if you use a label that starts with “L” or a *d* label (where *d* is a digit), as in this example

```
foo: b 1f
    ...
bar: ...
1:   ...
```

The resulting program won’t link or execute correctly, because only the symbols `foo` and `bar` make it into the object file’s symbol table. References to the temporary label `1` are compiled as offsets; as a result, no relocation entry is generated for the instruction `b 1f`. If the linker does not place the block associated with the symbol `bar` directly after that associated with `foo`, the branch to `1f` will not go to the correct place. Because there is no relocation entry, the linker doesn’t know to fix up the branch. The source-code change to fix this problem is to change the label `1` to a non temporary label (`bar1` for example). You can avoid problems with object files containing hand-coded assembly code by linking them whole, without reordering.

- the pseudo-symbol `.section_start`

If the specified section in any input file has a non-zero size and there is no symbol with the value of the beginning of its section, the linker uses the pseudo symbol `.section_start` as the symbol name it associates with the first block in the section. The purpose of this symbol is to deal with literal constants whose symbols do not persist into the object file. Because literal strings and floating-point constants are in literal sections, this not a problem for Apple compilers. You might see this symbol used by assembly-language programs or non-Apple compilers. However, you should not reorder such code and you should instead link the file whole, without reordering (see [“Linking with an Order File”](#) (page 21)).

Reducing Shared Memory Pages

As noted in [“Overview of the Mach-O Executable Format”](#) (page 9), the data in the `__DATA` segment of a Mach-O binary is writable and thus shareable (via copy-on-write). Writable data slows down paging performance in low-memory situations by increasing the number of pages that may need to be written to disk. For frameworks, writable data is shared initially but has the potential to be replicated to the memory space of each process.

Reducing the amount of dynamic or non-constant data in an executable can have a significant impact on performance, especially for frameworks. The following sections show you how to reduce the size of your executable's `__DATA` segment, and thus reduced the number of shared memory pages.

Declaring Data as `const`

The easiest way to make the `__DATA` segment smaller is to mark more globally scoped data as constant. Most of the time, it's easy to mark data as constant. For example, if you're never going to modify the elements in an array, you should include the `const` keyword in the array declaration, as shown here:

```
const int fibonacci_table[8] = {1, 1, 2, 3, 5, 8, 13, 21};
```

Remember to mark pointers as constant (when appropriate). In the following example, the strings "a" and "b" are constant, but the array pointer `foo` is not:

```
static const char *foo[] = {"a", "b"};
foo[1] = "c";           // OK: foo[1] is not constant.
```

To mark the entire declaration as constant, you need to add the `const` keyword to the pointer to make the pointer constant. In the following example, both the array and its contents are constant:

```
static const char *const foo[] = {"a", "b"};
foo[1] = "c";           // NOT OK: foo[1] is constant.
```

Sometimes you may want to rewrite your code to separate out the constant data. The following example contains an array of structures in which only one field is declared `const`. Because the entire array isn't declared `const`, it is stored in the `__DATA` segment.

```
struct {
    const char *imageName;
    UIImage *image;
} images[100] = {
    {"FooImage", nil},
    {"FooImage2", nil}
    // and so on
};
```

To store as much of this data as possible in the `__TEXT` segment, create two parallel arrays, one marked constant and one not:

```
const char *const imageNames[100] = { "FooImage", /* . . . */ };
NSImage *imageInstances[100] = { nil, /* . . . */ };
```

If an uninitialized data item contains pointers, the compiler can't store the item in the `__TEXT` segment. Strings end up in the `__TEXT` segment's `__cstring` section but the rest of the data item, including the pointers to the strings, ends up in the `__DATA` segment's `const` section. In the following example, `daytimeTable` would end up split between the `__TEXT` and `__DATA` segments, even though it's constant:

```
struct daytime {
    const int value;
    const char *const name;
};

const struct daytime daytimeTable[] = {
    {1, "dawn"},
    {2, "day"},
    {3, "dusk"},
    {4, "night"}
};
```

To place the whole array in the `__TEXT` segment, you must rewrite this structure so it uses a fixed-size char array instead of a string pointer, as shown in the following example:

```
struct daytime {
    const int value;
    const char name[6];
};

const struct daytime daytimeTable[] = {
    {1, {'d', 'a', 'w', 'n', '\0'}},
    {2, {'d', 'a', 'y', '\0'}},
    {3, {'d', 'u', 's', 'k', '\0'}},
    {4, {'n', 'i', 'g', 'h', 't', '\0'}}
};
```

Unfortunately, there's no good solution if the strings are of widely varying sizes, because this solution would leave a lot of unused space.

The array is split onto two segments because the compiler always stores constant strings in the `__TEXT` segment's `__cstring` section. If the compiler stored the rest of the array in the `__DATA` segment's `__data` section, it's possible that the strings and the pointers to the strings would end up on different pages. If that happened, the system would have to update the pointers to the strings with the new addresses, and it can't do that if the pointers are in the `__TEXT` segment because the `__TEXT` segment is marked read-only. So the pointers to the strings, and the rest of the array along with it, must be stored in the `const` section of the `__DATA` segment. The `__const` section is reserved for data declared `const` that couldn't be placed in the `__TEXT` segment.

Initializing Static Data

As is pointed out in [“Overview of the Mach-O Executable Format”](#) (page 9), the compiler stores uninitialized static data in the `__bss` section of the `__DATA` segment and stores initialized data in the `__data` section. If you have only a small amount of static data in the `__bss` section, you might want to consider moving it to the `__data` section instead. Storing data in two different sections increases the number of memory pages used by the executable, which in turn increases the potential for paging.

The goal of merging the `__bss` and `__data` sections is to reduce the number of memory pages used by your application. If moving data into the `__data` section increases the number of memory pages in that section, there is no benefit to this technique. In fact, adding to the pages in the `__data` section increases the amount of time spent reading and initializing that data at launch time.

Suppose you declare the following static variables:

```
static int x;
static short conv_table[128];
```

To move these variables into the `__data` section of your executable's `__DATA` segment, you would change the definition to the following:

```
static int x = 0;
static short conv_table[128] = {0};
```

Avoiding Tentative-Definition Symbols

The compiler puts any duplicate symbols it encounters in the `__common` section of the `__DATA` segment (see [“Overview of the Mach-O Executable Format”](#) (page 9)). The problem here is the same as with uninitialized static variables. If an executable's non-constant global data is distributed among several sections, it is more likely that this data will be on different memory pages; consequently, the pages may have to be swapped in and out separately. The goal for the `__common` section is the same as that for the `__bss` section: to eliminate it from your executable if you have a small amount of data in it.

A common source of a tentative-definition symbol is the definition of that symbol in a header file. Typically, headers declare a symbol but do not include the definition of that symbol; the definition is instead provided in an implementation file. But definitions appearing in header files can result in that code or data appearing in every implementation file that includes the header file. The solution to this problem is to ensure that header files contain only declarations, not definitions.

For functions, you would obviously declare a prototype for that function in your header file and put the definition of that function in your implementation file. For global variables and data structures, you should do something similar. Rather than defining the variable in your header file, define it in your implementation file and initialize it appropriately. Then, declare that variable in your header file, preceding the declaration with the `extern` keyword. This technique localizes the variable definition to one file while still allowing access to that variable from other files.

You can also get tentative-definition symbols when you accidentally import the same header file twice. To make sure you do not do this, include preprocessor directives to prohibit the inclusion of files that have already been included. Thus, in your header file, you would have the following code:

```
#ifndef MYHEADER_H
```

```
#define MYHEADER_H
// Your header file declarations. . .
#endif
```

Then when you want to include that header file, include it in the following way:

```
#ifndef MYHEADER_H
#include "MyHeader.h"
#endif
```

Analyzing Mach-O Executables

You have several tools at your disposal for finding out how much memory your non-constant data is occupying. These tools report on various aspects of data usage.

While your application or framework is running, use the `size` and `pagestuff` tools to see how big your various data sections are and which symbols they contain. Some things to look for include the following:

- To find executables with lots of non-constant data, check for files with large `__data` sections in the `__DATA` segment.
- Check for variables and symbols in the `__bss` and `__common` sections that can be removed or moved to the `__data` section.
- To locate data that, although declared constant, the compiler can't treat as constant, check for executables or object files with a `__const` section in the `__DATA` segment.

Some of the bigger consumers of memory in the `__DATA` segment are fixed-size global arrays initialized but not declared `const`. You can sometimes find these tables by searching your source code for `"[] = {"`.

You can also let the compiler help you find where arrays can be made constant. Put `const` in front of all the initialized arrays you suspect might be read-only and recompile. If an array is not truly read-only, it will not compile. Remove the offending `const` and retry.

Minimizing Your Exported Symbols

If your application or framework has a public interface, you should limit your exported symbols to those required for your interface. Exported symbols take up space in your executable file and should be minimized when possible. Not only does this reduce the size of your executable, it also reduces the amount of work done by the dynamic linker.

By default, Xcode exports all symbols from your project. You can use the information that follows to identify and eliminate those symbols you do not want to export.

Identifying Exported Symbols

To view the symbols exported by your application, use the `nm` tool. This tool reads an executable's symbol table and displays the symbol information you request. You can view all symbols or just the symbols from a specific segment of your executable code. For example, to display only the externally available global symbols, you would specify the `-g` option on the command line.

To view detailed symbol information, run `nm` with the `-m` option. The output from this option tells you the type of the symbol and whether it is external or local (non-external). For example, to view detailed symbol information for the TextEdit application, you would use `nm` as follows:

```
%cd /Applications/TextEdit.app/Contents/MacOS
% nm -m TextEdit
```

A portion of the resulting output might look like the following:

```
9005cea4 (prebound undefined [lazy bound]) external _abort (from libSystem)
9000a5c0 (prebound undefined [lazy bound]) external _atexit (from libSystem)
90009380 (prebound undefined [lazy bound]) external _calloc (from libSystem)
00018d14 (__DATA,__common) [referenced dynamically] external _catch_exception_raise
00018d18 (__DATA,__common) [referenced dynamically] external _catch_exception_raise_state
00018d1c (__DATA,__common) [referenced dynamically] external
_catch_exception_raise_state_identity
```

In this mode, `nm` displays various information depending on the symbol. For functions and other code residing in the `__TEXT` segment, `nm` displays prebinding information and the originating library. For information in the `__DATA` segment, `nm` displays the specific section of the symbol and its linkage. For all symbols, `nm` displays whether the symbol is external or local.

Limiting Your Exported Symbols

If you know the symbols you want to export from your project, you should create an exports file and add that file to your project's Linker settings. An exports file is a plain text file containing the names of the symbols you wish to make available to external callers. Each symbol must be listed on a separate line. Leading and trailing whitespace is not considered part of the symbol name. Lines starting with a `#` sign are ignored.

To include your exports file in your Xcode project, modify the target or build-style settings for your project. Set the value of the "Exported symbols file" setting to the name of your exports file. Xcode passes the appropriate options to the static linker.

To export a list of symbols from the command line, add the `-exported_symbols_list` option to your linker command. Rather than export a specific list of symbols, you can also export all symbols and then restrict a specific list. To restrict a specific list of symbols, use the `-unexported_symbols_list` option in your linker command.

Note that symbols exported by the runtime libraries must be included explicitly in your exports file for your application to launch properly. To gather a list of these symbols, link your code without an export file and then execute the `nm -m` command from Terminal. From the resulting output, gather any symbols that are marked `external` and are not part of your code and add them to your exports file.

Limiting Exports Using GCC 4.0

GCC 4.0 supports custom visibility attributes for individual symbols. In addition, the compiler provides compile-time flags that let you set the default visibility for all symbols for the compiled files.

For information on using the new symbol visibility features of GCC 4.0, see "Controlling Symbol Visibility" in *C++ Runtime Environment Programming Guide*.

Document Revision History

This table describes the changes to *Code Size Performance Guidelines*.

Date	Notes
2006-06-28	Added information about using GCC 4.0 to reduce the number of exported symbols.
2005-04-29	Clarified guidelines for inline functions. Updated compiler options to cover dead-code stripping.
	Document name changed. Old title was <i>Optimizing Your Code Footprint</i> .
2003-12-11	Minor bug fix to clarify use of the <code>-dynamic</code> compiler option.
2003-07-25	Minor bug fixes to reflect the Xcode environment.
2003-05-15	First revision of this programming topic. Some of the information appeared in the document <i>Inside Mac OS X: Performance</i> .

Index

Symbols

`__bss` section [10, 11, 33](#)
`__common` section [10, 11, 33](#)
`__const` section [9, 10, 31](#)
`__cstring` section [9](#)
`__data` section [10, 29](#)
`__DATA` segment [10](#)
`__dyld` section [10](#)
`__la_symbol_ptr` section [10](#)
`__nl_symbol_ptr` section [10](#)
`__picsymbol_stub` section [9](#)
`__text` section [9, 23](#)
`__TEXT` segment [9](#)
`__TEXT` segment [23](#)

A

ANSI C [11](#)
arrays, making constant [34](#)
assembly language, reordering code [30](#)

C

C++
 exception handling [15](#)
 uninitialized static data and [11](#)
code profiling
 with `gprof` [19](#)
cold routines [26](#)
compiler optimizations [13–14](#)
constant data [9](#)
copy-on-write [10](#)

D

dead code stripping [14](#)
declaring data `const` [31](#)
default order files [25](#)
duplicate symbols [33](#)
dynamic linker (`dyld`) [9](#)

E

exception handling overhead [15](#)

F

frameworks
 building [17](#)
functions
 inline [16](#)
 reordering [24](#)

G

`gprof` tool [19–22](#)

H

hot routines [26](#)

I

implementation files [33](#)
inline functions [16](#)

M

Mach-O

- executables [23](#)
- object files [23](#)

memory

- freeing [9](#)
- usage [9](#)

N

nonconstant data [10](#)

O

order files

- and literal sections [28](#)
- using [25](#)

P

pagestuff tool [25, 34](#)

paging [9](#)

Project Builder [24](#)

pseudo symbols [30](#)

R

read-only segments [9](#)

read/write data [10](#)

S

scatter loading [19](#)

sections

- __bss [10, 11, 33](#)
- __common [10, 11, 33](#)
- __const [9, 10](#)
- __cstring [9](#)
- __data [10](#)
- __dyld [10](#)
- __la_symbol_ptr [10](#)
- __nl_symbol_ptr [10](#)
- __picsymbol_stub [9](#)
- __text [9](#)

.section_start symbol [30](#)

-sectorder option [28](#)

size tool [34](#)

static data [33](#)

symbols

.section_start [30](#)

duplicate [33](#)

exporting [36](#)

identifying [35](#)

in runtime libraries [36](#)

removing [15](#)

T

tentative-definition symbols [33](#)

tools

gprof [19, 22](#)

pagestuff [25, 34](#)

U

unique utility [28](#)

W

warm routines [26](#)

working set [19](#)