



TALIGENT'S GUIDE TO DESIGNING PROGRAMS



WELL-MANNERED OBJECT-
ORIENTED DESIGN IN C++



Taligent®

Library of Congress Cataloging-in-Publication Data

Taligent's guide to designing programs : well-mannered object-oriented design in C++.

p. cm.

"Taligent Press."

Includes index.

ISBN 0-201-40888-0

1. Object-oriented programming (Computer science) 2. C++

(Computer program language) I. Taligent, Inc.

QA76.64.T34 1994

005.2--dc20

94-1186

CIP

Taligent and the Taligent logo are registered trademarks of Taligent, Inc.

All other Trademarks belong to their respective companies.

This manual and the software described in it are copyrighted. Under the copyright laws, this manual or the software may not be copied, in whole or part, without the prior written consent of Taligent.

The product described in this manual may be protected by one or more U.S. and International Patents.

The authors and publishers have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damages in connection with or arising out of the use of the information or programs contained herein.

Copyright © 1994 by Taligent, Inc. 10201 N. De Anza Blvd., Cupertino, California 95014-2233 U.S.A. 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, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

Printed in the United States of America. Published simultaneously in Canada.

RESTRICTED RIGHTS LEGEND: Use, duplication, or disclosure by the government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 and FAR 52.227-19.

Sponsoring Editor: Martha Steffen

Cover and text design: Taligent Technical Communications Group, Gary Ashcavi

Set in 10 point New Baskerville

3 4 5 6 7-CRS-9897969594

Third printing, October 1994

Addison-Wesley books are available for bulk purchases by corporations, institutions, and other organizations. For more information please contact the Corporate, Government and Special Sales Department at (800) 238-9682.

in an unbounded fashion, as the default stack always has some preset size. Algorithms that use recursion in a bounded fashion, such as Quicksort, are acceptable. The deeper you recurse, the less stack space you should use on each recursion.

Use the heap for larger objects, not the stack, regardless of performance. There are a number of custom storage allocation techniques you can use (such as pooling objects for reuse) to cut down on the overhead for large heap objects.

For information about memory issues with local arrays, see “Avoid using arrays as local variables or object fields” on page 53.

SHARED LIBRARIES

Shared libraries have many advantages, including easy software updates, code sharing, and the ability to dynamically load objects. However, they have one disadvantage: because the library is shared among many applications, unreferenced code and data can't be stripped. This is a minor problem for code, because it is immutable and is shared among all clients. Additionally, unused code doesn't get paged in, and the segmentation capabilities of the Taligent Linker segregate code into contiguous *chunks* that are used together to reduce fragmentation. Static data, however, is a problem.

Avoid static objects

Avoid modifying static data because the page it is on will no longer be shared. This is not an absolute rule; it's all right to have modifiable static data or static objects with constructors in a library, as discussed next.

The best way to avoid modifying static data is to not have it in a shared library. This includes static objects with constructors, because the constructors run after library initialization and modify the storage for the object, resulting in a separate copy for each task (plus anything else on the same page). Static objects or struct's initialized by C-style initializers can be shared between all tasks.

If you need static data that is modifiable or is an object, allocate it on demand. For example, rather than having a static array, have a static pointer, and allocate the array the first time it is needed. The same is true for objects with constructors; allocate them as needed. One useful trick is to place static objects inside functions rather than at file scope; then they are initialized the first time the function is called. This works for heap storage as well:

```
void TFoo::Bar()
{
    static TBaz *gWhatever = new TBaz();
    ...
}
```

CONCURRENCY AND SHARED LIBRARY ISSUES

There's an old proverb: "Be careful what you ask for—you might get it." This definitely applies to preemptive multitasking. It solves many problems, but like any panacea, it introduces new ones. The big problem for preemptive scheduling is concurrency and synchronization. If multiple threads are changing or reading data structures at the same time, the chance that the data structures will be consistent is about nil.

One approach is to require that every object in the system be safe for use by multiple concurrent threads. This works, but has a big performance impact and very little benefit. For example, consider an object A, which is built out of other objects B, C, D, and E, which are all safe for use by multiple threads. This does *not* mean that A is therefore safe for use by multiple threads. Although B, C, D, and E all separately preserve their individual invariants, that doesn't mean that A's invariants are preserved. In fact, in making A multithread safe, it is often the case that the components' being multithread safe is of little or no use, as it is A in its entirety which must be synchronized.

Therefore, synchronize high-level constructs only; avoid synchronization at a low level because it has storage and time penalties. There are some exceptions; Taligent's MReferenceCounted is multithread safe because it enables programming that is also multithread safe without any additional overhead (see "Surrogate objects" on page 91). However, the Collection classes are *not* multithread safe because you usually use them to construct other objects, rather than access them directly from multiple threads.

Synchronization techniques

The classic synchronization technique is to use some form of lock. There are two kinds of synchronization locks in the Taligent Application Environment:

- Monitors for synchronizing most data structures with multiple writers. Use monitors except when the semaphore model fits better.
- Semaphores for when there are many readers and one writer. There is less contention, because semaphores directly support multiple readers.

Another technique is to use surrogates (see "Surrogate objects" on page 91). If you can split an object into immutable shared data and per-thread mutable data, you can avoid some of the need for synchronization. For example, TGAreas are not shared between threads, but the master objects they point to are. Only the reference counts in the master objects need to be synchronized, because otherwise they do not change.

✓ NOTE Synchronization should almost always be performed inside an object, not by clients. Counting on clients to make acquire and release calls is asking for trouble, as they are sure to forget one or the other at some point.