Modular C

How to Write
Reusable and Maintainable Code
using the C Language.

Robert Strandh

1 Introduction

This is a guide to writing reusable and maintainable code using the C language.

Reusable code is essential to avoid duplicated effort in software development. Instead of rewriting software components from scratch, a programmer can make use of an existing component. In addition, reusable code helps performance by making it possible to put reusable software components in shared libraries. Thus, reusability is about avoiding duplicated effort between two or more different programs.

Maintainable code serves a somewhat different purpose. Here, the axis is time, but the program remains the same. Again, we try to avoid duplicated effort, but in this case within the same program undergoing evolution. To make such evolution easier, the code must be possible to understand and easy to modify with predictable results.

This guide is meant for the programmer who writes application programs using the C language. It is not meant for the C programmer doing system programming. System programming differs from application programming in that system programming emphasizes performance over reusable and maintainable code. Often these two goals are in conflict, but certainly not always.

There are many types of applications. This document is meant for applications that use symbolic programming, in contrast to numeric (number-crunching) and administrative (management information systems) programming. Symbolic programming refers to the manipulation of sometimes quite complex data objects and relations between them. Such objects typically represent objects in the real world such as people, motors, and regulators, or purely imaginary objects from computer science such as binary trees, file systems, and user accounts.

2 The C Language

The C language is not a very good language for writing applications, but it is not too bad. However, since the C language was initially meant to be a language for systems programming, it has numerous features that can make it much harder to reuse or maintain your program if exploited without restrictions. In addition, the C language contains some features that can actually be used to create good code, but these features are rarely used in practice. In this document, we try to point out features that should be avoided and propose alternative features.

There are some major advantages to using the C language for application programming. For one thing, it is relatively efficient. The quality of existing compilers is quite good, and a comfortable programming environment exists for Unix. With ANSI C, the compiler can verify the number and types of function arguments. The use of the keyword `static` makes it possible to encapsulate information. Interface and implementation can be clearly separated by using header files.

Unfortunately, there are major problems with C as an application language. Perhaps the most serious one is the absence of automatic memory management (garbage collection). The lack of automatic memory management has a tendency to penetrate abstraction barriers and expose implementation details to clients of a module. Furthermore, the lack of automatic memory management in standard C prompts most programmers to try to manage memory in ad hoc ways that create long-term maintenance problems. For example, in a program that requires a great deal of dynamic memory allocation, programmers are tempted to use a great many pointers to objects for the sake of efficiency. However, having a great many pointers to a given object makes it hard (if not impossible) to tell when it's safe to de-allocate that object. To cope with that uncertainty, some programmers resort to copying the object, so that they can be reasonably sure they are the only one using it, so that they can safely de-allocate it when through. This copying may address the problem of too many dangling pointers, but it has the unfortunate effect that objects are no longer being shared. The direct of effect of no longer sharing objects but, rather, relying on copies of them, is that there is far greater possibility of inconsistency between copies of an object—surely a situation we hope to avoid.

Given that ANSI C suffers these limitations as an application language, this guide still offers some helpful principles toward writing better C programs.

3 Reusability and Maintainability

3.1 Reusability

Reusability refers to the quality of a software component that is sufficiently general that it is independent of the current programming problem. Since it is independent it could in principle be used in a different program as well. But independence is not sufficient in order for a component to be reusable. It must be sufficiently easy to understand how to use the component so that writing a new one from scratch is simply not worthwhile. Among other things, a component with many competing purposes will be more difficult for maintainers to understand than will a single-purpose component. It follows, then, that the component should have a very specific purpose. This purpose can be to implement either a function or a type.

During the days of structured programming, the purpose of a components was always to perform a particular function. This method works reasonably well for numeric problems, mostly because the types of manipulated data are quite simple and standardized (integers, floats, complex numbers, and arrays of these).

In symbolic programming, however, the data objects tend to be much more complex than that. Typically, objects do not behave exactly the same way in two different applications. It is thus necessary to allow for parameters to adapt components to different applications. For this type of programming, it seems more appropriate to create components based on the type of the objects manipulated. Thus, a component would usually contain the definition of an object type and all the operations that are possible on objects of that type. Throughout this document, we are going to use the word "module" to refer to such a component. Modular programming is the key to reusability.

As far as reusability is concerned, the way a particular module was implemented does not have any impact. Only the interface of the module is needed in order to reuse it. Thus, reusability concerns only the way the module is presented to the outside world.

3.2 Maintainability

Maintainability refers to the facility with which a program can evolve. It has been estimated that more than three-quarters of the total time spent on a particular program is evolution and other modifications, such as fixing bugs. It therefore becomes necessary to write the program not only so that it works in the first place, but also in order to make it easier to modify to produce new, improved versions.

Usually, the person maintaining a particular program is not the same person that wrote it. It is therefore of great importance that the code be written in a way that it can be understood by someone else and modified with predictable results.

While modularity is the key to reusability, and indeed necessary in order for a program to be maintainable as well, it is not enough. The implementation of a module is susceptible to evolution as well. Therefore, maintainability is determined by the way a module was implemented in addition to its interface.

4 How to Write Reusable Code

The main purpose of this document is to communicate to the programmer a way of thinking when programming in C. Writing reusable code is a state of mind. You constantly have to be aware of the consequences of what you write. Acquiring this skill may take years of active programming, and can certainly not be understood only from a presentation like this one.

It is tempting to try to make a list of requirements for a reusable module. Such a list is necessarily incomplete. Furthermore, it is not possible to take such a list and understand from it what to do in a particular programming project. We nevertheless present such a list, so that the programmer can use it as a checklist to verify that the code that was written is indeed reusable. That checklist appears in See Chapter 7 [cookbook], page 26.

4.1 Functional Modules or Data Modules

There are two different ways of making a module. The first way is for the module to implement a certain function, not in the mathematical sense, but in the sense of a functionality, the ability to accomplish a task. The second way is for a module to contain all the operations on a certain type of object. In that case, we say that the module is an implementation of an abstract data type.

The two different kinds of modules result from two different methods for dividing the application into smaller pieces.

Top-down design was made popular during the days of structured programming. The idea is to start by deciding what the application does. The short description of what the application does is then refined into parts that take care of different subtasks, and so on until low-level code is obtained. The modules obtained by this method are typically functional, i.e., the purpose of a module is to take care of a particular functionality of the application. Functional modules obtained by top-down design tend to be particular to the application in question. This particularity is a direct result of the way they are defined: they implement a part of the main goal, namely, what the application does.

Bottom-up design is a more modern way of creating modules. The idea is to put off the decision about what the application does as long as possible. Instead, we start by defining the types of objects that the application is likely to manipulate. We call these types “abstract data types.” The next step is to define operations on the abstract data types, independently of how they are going to be used in the application. This way, we obtain modules that correspond to data instead of to functions.

4.1.1 Functional Modules

Top-down design and stepwise refinement encourage modules that implement functions. Most traditional design methods generate modules that implement functions.

The problem with modules that implement functions is that several modules must operate on the same types of data. Therefore, it is not possible to hide the data definitions inside a module. Since data definitions tend to reflect implementation details as well as

abstract data types, we must make implementation details globally visible, a practice that hinders both reusability and maintainability.

Let us say, for instance, that in a windowing system we have a module that can resize windows and a module that can move windows. These are clearly functions of a window system. Since these functions operate on the internals of a data structure that represents a window, they must both know the details of this data type. Thus, we are obliged to put this data definition in a header file to be included in the two modules. But once we put the data definition in a header file, nothing prevents a user from including it as well. We have now made public details of the implementation of a window, i.e., since the data definition, and thus the header file, contains parts that concern only implementation details (for instance, the size and storage type of the position and dimensions of a window) we have now exposed these implementation details to anyone who cares to use them, with the obvious unhappy repercussions for reusability and maintenance.

4.1.2 Data Modules

The other way of making a module is to concentrate on data. The module contains all the operation on a particular data type. For instance, if "person" is a data type, then the module that implements the data type person contains operations such as computing the date of birth or the address of the person.

Data modules are capable of hiding implementation details. The only part of the module that is visible to a client is the interface, i.e., the operations. The exact implementation is not visible, and can thus be modified without affecting the client.

4.2 Interface vs. Implementation

A reusable module contains two distinct parts: the interface and the implementation.

The interface describes to a client how this module is supposed to be used. In other words, if the module is a data module, it describes the operations allowed on an object of the type defined by the module. These operations should be described in an abstract way, without referring to the implementation of the module. In fact, it should be possible to change the implementation without changing the interface. For instance, let us suppose we are writing a module for the type "set." The operations on sets would be, for instance, create an empty set, create a singleton set, take the union of two sets, take the intersection of two sets. The interface should then still be valid even if two completely different implementations were chosen, whether as a hash table or a linked list, for example.

In the C language, we should try as much as possible to use the header file as the place to put information about the interface, and the C source file as the place to put implementation details. In particular, if the data type is implemented as a C structure, the declaration of that structure should be in the `.c` file. Putting structure declarations in the `.h` file usually means exposing implementation details to the client of a module, which is contrary to modular programming.

Why do we object so strongly to exposing implementation details to a client? To answer that question, we'll compare the implementation of a module to tactics, and the interface of a module to strategy. Just as tactics need to remain fluid, flexible, and subject to change,

likewise an implementation may evolve, and as it evolves, we want to disturb clients of the module as little as possible. The interface, in contrast, like strategy, should remain stable over the long haul. It should be something that the client can count on using with as little change as possible.

4.3 Functional vs. Imperative Interface and Implementation

There are essentially two ways of specifying the operations on an abstract data type.

In one style, the operations behave like functions, i.e., they take arguments and return a newly constructed object without modifying the arguments. For instance, if the operation is to take the union between two sets, then with a functional interface, the union operation would return a third set which is the union of its arguments, without modifying any of its arguments. This style of interface is popular in the functional programming community, as it is a prerequisite for referential transparency, which makes it easier to prove properties about the program.

The other interface style is the imperative style. In this style, operations do not return any new values. Instead, operations behave like procedures, i.e., one or more of the arguments is modified by the operation. For instance, suppose again we wish to define the union of two sets, but this time with an imperative interface. Then the union operation would be declared to return `void`, and the operation would be defined to modify (say) the first argument so that after the operation the first argument would contain the union of the initial first argument and the second argument. The second argument would not be modified.

Whether you choose a functional or imperative interface, the implementation must be faithful to that interface. If a functional interface has been chosen, the implementation is not allowed to modify any of the arguments. This restriction may imply the necessity of allocating new memory for the return value. If that is the case, memory de-allocation is a bit of a problem. In the presence of shared objects, a clear protocol must be established about two issues: which module is responsible for de-allocation, and when such de-allocation should be done. Another possibility would be to use a garbage collector, such as `cgc`.

If, instead, an imperative interface has been chosen, the implementation must modify the object as indicated. This can be problematic in the presence of shared objects. For instance, if two variables both contain the same object through (say) assignment of one to the other, then if one of the variables is given to an operation that modifies it, then the modification must be reflected in the second one. Thus, if both `x` and `y` contain some kind of set of elements, applying the union operation to `x` and some third set, must preserve the equality of `x` and `y`. This restriction makes it somewhat tricky to implement certain data types. It excludes, for example, representing the empty set as a `NULL` pointer. Here is why: If `x` is assigned to `y` which contains the empty set, and then the `y` is used in a union operation with a nonempty set, our restriction requires that `x` and `y` be equal after the operation, but there is no way for the union operation to modify the value of `x`.

The solution we propose to this problem is for the implementation to use headers for the objects that are to be represented. Thus, even the empty set, the empty list, etc. would be represented as a header object that would be created by the operation that creates an empty object. The real set or list would then be contained in the header object. This solution has

the additional advantage that the header can be used to hold additional information about the object, such as the length of the list, the number of objects in the set, the comparison function for elements, etc.

Although this solution may seem like a waste of memory, it is necessary in order to hide implementation details. Solutions that make it possible for a client to conduct an experiment that reveals the details of the implementation must be avoided at all cost. Otherwise, one can be certain that a client will exploit this knowledge and as a consequence compromise the maintainability and reusability of the module.

4.4 Absence of Arbitrary Limitations

It is tempting to limit certain possible sizes of data types. For instance, one might argue that 256 characters is enough for a file name since Berkeley Unix imposes this limitation. Or, one may think that the length of a line of text could be restricted to 1024 characters because nobody would type lines that are longer than this.

However, assuming that a module is to be used in a certain way is usually a bad idea. A new version of Unix may get rid of the limitation on file names. Then the module may have to be rewritten. Worse, nobody notices and one day a program that uses longer file names simply crashes or gives erroneous results. Or someone may use the module for lines of text that have not been typed by humans, but generated by some other program, and that are not intended for humans to read. Such lines may very well be much longer than 1024 characters.

A fundamental principle of reusability is thus to avoid arbitrary limitations of all kinds. Limitations must, of course, exist. The trick is to avoid the arbitrary ones. A limitation of the length of a file name to 2**32 is not serious and not arbitrary, since that may well be the limitation of addressability of the machine architecture.

4.5 Few Assumptions

The module should make as few assumptions as possible. It is a common mistake by implementors of modules to assume that their module is going to be used in a certain way.

For instance, it may be tempting to write a hash function for strings that only looks at the first two letters of the string. The assumption being that the strings are random and therefore are likely to differ in the first two characters. However, the hash function may be used on names of a particular library in which the functions have systematic naming conventions, for instance by a constant prefix on each function. In that case the assumption is no longer valid and the result is a severe performance problem.

Another example is to assume that a module is going to be used to read text files, and thus not to treat the case were other characters are included. An example of this practice can be seen in the lexical analysers generated by the Unix `lex` utility. Such a lexical analyser uses the value 0 to indicate end of input. Thus, it is impossible to analys binary input that naturally contains 0 valued bytes.

If an algorithm is more complex than necessary, it may pose a soft but very serious limitation. No hard limit is imposed, but the module is essentially useless for data sizes

above a certain limit. For instance, in a module that provides a way to search a text for a certain occurrence of another text or a pattern, a straightforward implementation would be to compare the pattern to the text at every position possible. The running time of such an algorithm is proportional to the product of the size of the text and the pattern. For large texts and patterns this may be unacceptable, in particular since there are linear algorithms to do the same things.

The solution is for the module writer to avoid such assumptions if possible. Things to watch out for:

Assuming Unix directories with limited length file names

Assuming text file input

Assuming fixed line lengths

Assuming small sizes of objects

Assuming a small number of objects

4.6 Few Parameters

For a module to be reusable, its parameters should be straightforward to specify. There might be a temptation to provide many parameters so that a user can optimize the module for specific purposes. However, too many parameters may make a user reluctant to invest the effort in learning what they mean. The result might be that parameters are specified in a nonoptimal way and performance is decreased. In the worst case, the potential user will decide to write a special purpose module that is easier to understand, but in the rewriting, wastes valuable programming effort.

For instance, a module for hash tables might have a parameter to determine the size of the hash table. However, a potential user may not know what a good value would be. Setting it too low causes performance penalty if the user's guess is wrong. Setting it too high wastes memory.

The solution is to make it possible to resize the hash table dynamically. A small value is given for the size of the empty hash table, and the size should grow as necessary. Under those conditions, the user does not have to specify the size, yet performance remains good.

4.7 Precompilation of Module

The module should be made in such a way that it can be precompiled and installed in a library without the module needing access to client code. This practice allows the code to be shared between several applications. It also avoids unnecessary recompilations when client code changes.

For example, a module may have a parameter that determines the size of a data structure. A client module would specify this parameter as a configuration before the module is compiled. Thus, the module cannot be precompiled without the module knowing the value of the parameter, and the module must be recompiled for each client and each time the size parameter changes.

The solution is to avoid compile-time parameters.

4.8 Genericity vs. Polymorphism

Some modules define data types that contain other data types. Such modules are called containers. Examples of containers are stacks, lists, queues, hash tables, trees, etc. There are essentially two different ways of specifying the type of the objects contained. These two ways are referred to as genericity and polymorphism. The C language supports neither very well, but it is possible to obtain either by clever use of the C preprocessor and generic pointer types.

4.8.1 Genericity

A container module is called generic if it is written so that the exact type of the objects contained is not determined when the module is written. Instead, the type is specified in the form of a type variable. The value of the type variable is instantiated at compile time by a client module that uses the container. For instance, a client needs a binary tree of integers. The client can take a generic binary tree module and instantiate the type variable to be integer. After instantiation, the generic module becomes an ordinary module that must be compiled in the normal way.

Generic modules must be instantiated at compile time. Each instance is a module with slightly different code. Thus, generic modules cannot be shared at runtime. A generic module can not even be precompiled and put into a library for later use. There always remains a part of the compilation to be done after instantiation.

The reason for instantiation at compile time has to do with the size of objects. The idea of a generic module is to be able to store objects of any size. Thus, for instance, to move or allocate objects, the size needs to be known. The code for moving objects may depend on the size of the objects. Thus code generation must be deferred until the type is known. Since the final code is not known until the client code exists, the final compilation cannot be done until client code is known. On the other hand, the linker is not sufficiently flexible to allow for code generation at link time. Thus, all code generation must be done before linking the program. For that reason, generic modules are instantiated in a compiler pass between compilation and linking.

Generic modules violate the principle that a module should not depend on client code, and that a module should be possible to precompile and put in a shared library.

The way to obtain genericity in C is to use the C preprocessor to replace type variables by real types before compilation. Doing that is not easy, especially in a way that name conflicts are avoided and that encapsulations of private data are maintained.

4.8.2 Polymorphism

Another way of obtaining modules that can be used with different data types is to use polymorphism. A polymorphic module is written so that it can be compiled and put in a library while still being able to accept objects of different types.

For a module to be polymorphic, all possible objects contained in it must be of the same size. It is possible to avoid this limitation by passing the size of the objects as a parameter to the module, but we recommend this practice only in exceptional cases. For instance, the

qsort function is a polymorphic module. It accepts arrays of any type of object, and sorts the objects according to a comparison function that was passed as an argument.

If the size is not to be passed as an argument to the module functions, then all objects must be of the same size. The only way to accomplish that is to make all objects the size of a pointer. This practice is known as pointer abstraction.

4.9 Pointer Abstraction

If we systematically manipulate an object indirectly by a pointer, we say that we use pointer abstraction. Pointer abstraction has many advantages as far as reusability is concerned. First, it allows us to hide implementation details of the object. All operations on the object must go through functions that take a pointer as an argument. Second, it is more efficient to move pointers around than to move potentially large objects. Third, it allows for containers to be polymorphic, i.e., since all objects are the size of a pointer, we can create container modules that manipulate objects of any type. Finally, since an object exists in one and only one copy, we can be sure that side effects to one reference are reflected in all of the references.

With pointer abstraction, since an object is only referred to through its pointer, there is no need for client code to have separate type names for the pointer and that to which it points. Client code may consider that the pointer IS the object. We do this by the declaration

```
typedef struct datatype *datatype;
```

in the C header file. Client code then manipulates the pointer type exclusively.

The main disadvantage of using pointer abstraction uniformly is that it usually requires more dynamically allocated memory from the heap. Using a great many objects allocated on the heap can pose a problem in C, since it normally does not offer a garbage collector. Thus, in managing these allocated objects without the assistance of a garbage collector, we have to be careful not to de-allocate an object that may be pointed to by other parts of the system. One widely adopted strategy for coping with this problem is to keep reference counters for objects. Otherwise, knowing exactly how many pointers refer to an object may be hard. Another possibility is to use the `cgc` garbage collector for the C language.

Another inconvenience with pointer abstraction is that container modules are no longer type-safe. A container module, in fact, converts the objects that it contains to generic pointer types. As a consequence of that conversion, client code must use type casts to have access to its own data type implementations.

4.10 Data-driven Programming

Sometimes we can greatly facilitate the encapsulation of private data by letting functions be determined by data. This style of programming is known as data-driven programming, and is best explained by an example.

Imagine, for instance, a windowing system that contains many different types of objects. An event loop is responsible for receiving external events and, according to the object type concerned, the event loop should call a particular function associated with the object type.

The most natural way to solve the problem is to put a `switch` statement inside the event loop. The labels of the switch statement would be type codes for objects, and the code for each case would be a call to the function that knows how to handle that particular type.

The problem with this solution is that the module in which the event loop is located must know all possible types in the system so that all the labels can be enumerated. Thus, the module containing the event loop must clearly be modified each time an object type is added, removed, renamed, or otherwise modified. Clearly, this is not a good solution.

A better solution is to have a globally known hash table that contains the functions associated with each object type and indexed by the type code given by the external event. Now, the event loop contains no information about the available types. All it does is take the type code, looks it up in the hash table, where it finds a function that it calls.

Clearly we have associated a function with some kind of data. In this case, the function to handle a particular object type was associated with the type code by means of a hash table.

Another similar example is a command interpreter for some kind of interactive system, such as a debugger. Generally, a module would handle a particular command or set of commands, for instance inserting or deleting a break point. Now, the command loop could either contain a big if-then-else statement that determines the command that was given and then calls the operation in the module concerned, or it could maintain a hash table that contains the correspondence between character strings (i.e., the command names) and the functions that handle the commands.

In the former case, the command loop must be modified each time a new command is added. In the latter case the command loop does not change at all.

In general, large switch statements indicate a potential problem with respect to reusability and maintainability (mostly maintainability, actually).

4.11 Programming by Contract

In the days of structured programming, it was believed that a module should be impossible to break, i.e., it should try to detect every possible error situation and report it. This style of programming is now referred to as defensive programming.

But defensive programming makes it harder to understand the code, simply because it tends to be swamped by error detections that hide its main purpose. In addition, this style is very inefficient, because error situations are tested for even though they are known not to be possible, perhaps because they were already tested for in some other module.

Today, we prefer to use a style that is called programming by contract. Instead of testing for all possible error situations, each module makes a programming contract. The contract clearly states the obligations of client code, i.e., what conditions must hold in order for the module to produce the desired result. If these conditions are fulfilled, the module has a contractual obligation that is also clearly stated. However, if client code does not respect the contract, the module has no obligations.

For instance, suppose that a list module contains an operation to take the first element of the list. A typical contract would require client code to supply a nonempty list. The module supplies a separate operation for testing whether the list is empty or not, but client

code may elect not to use it, perhaps because it just put an element on the list, and it thus knows that the list cannot be empty. For the module to test for emptiness in this operation would be a waste of time in the case where client code can prove that the list cannot be empty. In addition, the module code would be cluttered by the test for emptyness and the real purpose of the operation would be somewhat less clear.

An excellent way of making the terms of the contract explicit is to use `assert` clauses. An `assert` clause evaluates a statement. If the statement is true, then execution continues; if the statement is false, execution halts, and the application exits with a nonzero exit status.

Using `assert` clauses amounts to testing conditions, but by the fact that it is an `assert`, we clearly communicate to a reader of the code (a maintainer, for example) that the test has to do with the programming contract. Furthermore, in the context of an `assert`, we know that the program will terminate its execution whenever there is a breach of contract anywhere. Finally, when we think the program works, we can compile it with a flag that turns the `assert`s into `no-op`s so that no runtime penalty is induced while we still preserve the explicit statements of the contract.

4.12 Error Situations

One of the more difficult situations to handle is that of errors. The main problem lies in the fact that one has a tendency to mix up different kinds of errors. Some errors are so serious that immediate termination of the application is warranted (for instance, to avoid irreparable contamination of a database) whereas other errors could simply result in a warning to the user. Also the exact same error can lead to different consequences in different applications. For example, attempting to open a file that does not exist has different implications in a text editor than in a process control system. For those reasons, the way an error should be handled depends on the programming contract.

Roughly, we can divide errors into two kinds: fatal and nonfatal. Fatal errors are those that are so serious that the application should exit. Nonfatal errors are those where steps can be taken to recover; they can usually be handled by `exception`s.

4.12.1 Fatal Errors

As we indicated, fatal errors are those that make an application terminate.

Whether a certain error is fatal or not depends on the application. For instance, an attempt to open a file that does not exist could be fatal in a batch application such as `grep`. There is no point in continuing the execution of `grep` if there is no input file available. The same is not true for an interactive program such as `Emacs`. An attempt to open a file that does not exist should simply generate a message and a possibility for the user to continue with his or her task.

Notice, however, that opening a file in an interactive program is fatal if it is an installation file that must exist in order for the program to execute. Nonexistence of that file indicates an incorrect installation.

4.12.1.1 Contract Breaches

Breaking a programming contract induces a fatal error that indicates that the program contains a defect. Continuing execution of such a program can lead to disasters, as the state would be unknown. Execution should be terminated immediately. Incidentally, there is no point in trying to make the error message understandable to the user, since there is nothing the average user can do about it. An `assert` clause is an excellent way of testing for this kind of error situation.

4.12.1.2 Contractual Fatal Errors

Let's consider an example to clarify how a contractual fatal error differs from a breach of contract.

If a batch program tries to open a file that does not exist, (for example, if a user has typed `grep abc filename` but `filename` does not exist) usually a fatal error occurs, but that attempt is is not really a breach of contract. The user probably mistyped a file name and should be given the possibility of correcting the mistake and restarting the program. Thus, it is part of the contract to detect and report such errors. Errors of this kind are not really well handled by `assert`. These errors are best handled by a central error routine called `fatal`, which prints an error message and then calls `exit` with a nonzero exit code.

In order to detect and report this kind of error, client code must sometimes create an additional abstraction layer on top of certain reusable modules. A reusable module for maintaining files may, for instance, state that in order for client code to open a file, the file must exist. If client code calls this module directly as a result of the user typing a certain file name, clearly a nonexisting file name will provoke a breach of contract. Thus, client code must first verify, probably by some other operation provided by the same reusable module, that the file exists. If not, a call to `fatal` is made; otherwise, a call to the `open` operation is made.

4.12.2 Nonfatal Errors

4.13 Exceptions

Some people confuse exceptions with errors, but not all exceptions are errors, and not all errors produce exceptions. In fact, exceptions may or may not indicate error situations. Exceptions are simply control structures that allow the normal flow of a program to be altered. In some respects, exceptions are nonlocal `goto` instructions. They can be used to treat errors or simply to make the program more readable by avoiding numerous tests in the normal program flow. Generally, exceptions are used when the particular situation handled by the exception is rare compared to the normal control flow of the program.

A major use for exceptions is in interactive applications with a command loop. Generally, there may be many function calls between the command loop and a leaf routine that detects an exceptional situation. For instance, in a text editor, the command to read a file into a buffer may generate a call to the command interpreter, then to several intermediate

functions before finally calling the function that opens the file. If the file does not exist, we have an exceptional situation. A good way of treating this situation is for the leaf routine to generate an exception that is caught by the command interpreter, which restarts itself. This way, intermediate functions need not be aware of potential errors and can thus be written in a clearer way.

The more traditional way of treating this type of situation was made popular by the FORTRAN programming language and by the first versions of Turbo Pascal, both of which lack a nonlocal `goto` instruction, making it impossible to create an exception facility. In this programming style, all functions return codes that indicate errors. To propagate error codes under those conditions, intermediate functions must test error situations that they normally should not have to care about. It is not uncommon for code size to double as a result of this programming style. Needless to say, this style is highly undesirable and should be avoided.

In the C language, we use the library routines `setjmp` and `longjmp` to handle exceptions. We can use them either directly or indirectly (through an exception mechanism created on top of `setjmp` and `longjmp`). In our example of the text editor, the command interpreter would save the context using the `setjmp` function. When the leaf routine detects an error, it makes a call to an error function that calls `longjmp` to restore the context of the command interpreter. To avoid having the variable that contains the execution context be globally visible, we can put the error function in the same module as the command interpreter.

5 How to Write Maintainable Code

There are two aspects to maintainable code. Each module of the application must be written so that it is maintainable, and the combination of the different modules that form the application must also be maintainable. However, if the rules of reusability have been respected, then the combination of various modules should be straightforward, so the main problem that remains has to do with the maintainability of each module. Thus, in this chapter we concentrate our effort on the `.c` file that contains the code for a single module.

5.1 The Open-Closed Principle

This principle is described in an excellent way by Meyer []. The main idea is that a module must be both open and closed, as we'll explain.

A module is said to be closed if it has a well defined, complete interface. An interface is complete if no other operation is necessary in order to use the module. Thus, the module is considered closed if its interface is completely determined and is not going to change in the future. Such a module can be used by client code with no risk of incompatible future changes.

A module is said to be open if new operations can be added to it in a backwardly compatible way. None of the old operations are modified, but new ones can be added without breaking the old interface. Thus, the module is open if it supports the addition of new operations.

In order for a module to be maintainable, it must be both open and closed. There is no contradiction here, since the definitions of open and closed are not mutually exclusive. Thus, it is important for a module to be complete, but still allow new compatible operations to be added.

5.2 Naming Conventions

In this section, we'll indicate some popular naming conventions that should be avoided, and we'll offer some guidelines for names that ease maintenance headaches.

5.2.1 Hungarian Naming Conventions (a bad idea)

Hungarian naming conventions prefix variable names by a code that gives an indication of its type. For instance, all integers are prefixed by the letter `i`, all pointers by `p`, etc.

The hungarian naming convention—for which Hungarians should not be held accountable—represents an unfortunate attempt to set some conventions for coding. What is really obtained by such naming conventions is total breakdown of abstraction barriers. One of the fundamental principles of software engineering is to hide from a client all information that does not concern that client. Prefixing a variable by its type exposes exactly these unimportant implementation details to client code. Now, if the type of the implementation of a data type changes, all of the names of instances of that type must be altered as well.

5.2.2 Prefixing Fields by the Structure Name (a bad idea)

Prefixing fields of a structure by some abbreviated part of the name of the structure is not only unnecessary, but directly duplicates the name information in all the fields. A change of the name implies changing all the field names as well, in addition to all references to a field name. Surely this is a maintenance nightmare that we don't want to perpetuate.

5.2.3 Module Prefix

A useful naming convention for the parts of a module that are exported and thus globally visible (usually the interface functions), is to prefix the names with the name of the module, or some recognizable part of its name. For instance, a module that implements a data type "set," could have its interface functions prefixed with `set_`. This is an easy way to avoid conflicts between modules—conflicts about common names such as `insert`, `delete`, etc.

Notice that suffixing is less useful, since some linkers limit the number of significant characters that they can handle.

For names that are locally known to a module, there is no need to use a prefix. Indeed, it could be useful for different modules to use the same name for internal functions with a similar purpose. For instance, the name `init`, may be used inside a module to initialize it. For programs using the `cgc` garbage collector, a locally known function to copy an instance between the two memory halfspaces could be called `copy` in all modules.

5.2.4 Traditional Naming

Naming conventions should take traditions into account. For instance, the `main()` function of a C program takes three arguments describing the number of command line arguments, a table of these arguments, and a pointer to an environment, respectively. These arguments are traditionally called `argc`, `argv`, and `envp`. Not only is it not useful to use different names, it is highly desirable that exactly these names be used so that programmers recognize them immediately.

Similarly, the variable name `i` and `j` are always loop counters. Never use these variable for other purposes, and always use these variables first for loop counters.

It is important never to mislead the maintainer by using incorrect names. An example of bad practice is the use of the identifier `fd` (usually a general-purpose file descriptor) for an object of type `FILE *`. Now, a file descriptor is defined in Unix to be an integer that is returned by the operation `open()`, and used as an argument to operations such as `read()` and `write()`. Distinct from it, there is also the concept of a file pointer `FILE *`. This is the type return by `fopen()` and used in operations such as `fprintf()`, etc. But a file pointer is not a file descriptor. Therefore, using `fd` as a variable of type `FILE *` is highly misleading to the maintainer.

5.2.5 Short Names and Long Names

Usually long names are more descriptive than short ones, but names can be descriptive for different reasons than the number of characters. For instance, the variables named `i`

and `j` are nearly always used as loop counters. Calling these variables `loop_count_one` and `loop_count_two` or something similar is definitely not a way of making the program more readable.

Similarly, it is sometimes reasonable to call variables `x` or `y`. For instance, as arguments to a graphics routine that uses them to determine the position on the screen for some kind of object, these humble variables are well named. Similarly, for a function that implements arbitrary precision multiplication of two numbers, there is no reason to call the arguments `multiplicand` and `multiplier`. Simply use `x` and `y`!

Short names can also be used where the visibility of the variable is very limited. For instance, in order to traverse a list using a temporary pointer variable, we might use the name `p` for this variable as in the following example:

```
{
   struct list *p;
   for(p = first; p; p = p -> next)
      {
         /* only a few lines */
      }
}
```

Using a longer name may cause the line

```
for(p = first; p; p = p -> next)
```

to require more than 70 or so characters, making it hard to read. We may then have to split it into several lines, which increases the code size, i.e., the source, and creates a less common way of indenting the `for` loop.

5.3 Indentation

Correct indentation is essential for maintenance. Indentation should reflect program structure and should be consistent. The exact style of indentation used is of less importance than consistency and structure.

5.4 Visibility and Scope

One of the main principles of maintainability is to limit as much as possible the scope or the visibility of names of variables, functions and other programming elements.

A function defined in a module as part of the interface of that module must, of course, be visible to client code. But functions that are used internally should be invisible to client code. In C, we do this by using the key word `static` in the definition of the function. Using `static` is a simple and powerful way to communicate to the maintainer that there is no need to look for users of this function outside this module.

Sometimes it is necessary for a module to preserve some state between two calls to possibly different interface functions. Such a state must normally be contained in global variables. We do not, however, want to make such state accessible to client code. We can therefore use the key word `static` to hide such variables from client code. If the state is shared between two calls to one and the same interface function, we can use the `static` key word on a variable defined locally in the function. This practice limits the visibility of the variable to the body of the function—better than visibility in the entire file.

Variables should be defined as late as possible to limit their visibility. For instance, if a variable is shared between the last two functions defined in the file, we want to put the definition right before the first function that uses it, rather than, for instance, defining all global variables at the beginning of the file. Automatic variables should also be defined as late as possible. For instance, if we have a nested loop such as:

```
for(i = 0; i < m; i++)
  {
     ...
     for(j = 0; j < n; j++)
       {
          ...
       }
  }
```

Then we should put the definition of the variable `j` inside the first loop in the following way:

```
for(i = 0; i < m; i++)
  {
     int j;
     ...
     for(j = 0; j < n; j++)
       {
          ...
       }
  }
```

rather than (say) at the beginning of the function containing the loop. This is a simple way of limiting the visibility of variables.

5.5 Normalization

Normalization is a term that we have borrowed from database theory, and which we define to mean absence of duplication of information. If information is duplicated, it must be updated in every place when modified—clearly a bad idea with respect to maintenance.

Duplication can occur unfortunately in several different ways, such as duplicated pieces of code, duplication of information between code and comments, between code and documentation, and so on. In the sections that follow, we'll explain how to avoid those pitfalls

Normalization is enforced by the use of abstractions. Abstraction, in this context, means giving a meaningful name to a section of data definition or a section of code. The name should represent some useful entity that has meaning to the programmer.

5.5.1 Duplication of Code

Duplication of code occurs when the same piece of code, perhaps slightly modified, exists in several different places in the program. In other words, there is repetition of large chunks of code in the source. Such duplication should be avoided, for reasons of code size but more so for reasons of maintainability.

In contrast, for instance, a macro that creates a copy of a big chunk of code whenever instantiated may create a huge executable program, but that macro is no problem with

respect to maintainability, as the body of the macro exists in only one place. Thus, there is no unnecessary, unmaintainable repetition.

However, overuse of cut and paste in the text editor can easily create duplicated code in the sense that we're warning against. Not only does this kind of repetition tend to fill up the screen, making it harder to see the structure of the program, but this kind of repetition also makes it necessary to find all the instances whenever some modification is necessary.

We solve this problem of repeated chuncks of source code by liberal use of functional abstraction; i.e., whenever the same piece of code appears twice with no or only slight modification, we try to create a function that contains the code. The two occurrences are replaced by calls to the function, possibly with arguments accounting for the slight difference. Of course, it is not a good idea to create a function from an arbitrary sequence of instructions that happen to be similar. Rather, we try to identify pieces of code that have a specific purpose. We then assign a descriptive name to the piece of code in the form of a function.

Sometimes it is impossible to use functions to describe common pieces of code. In that case, we may use a macro. For instance, if the code contains a `goto` statement the target of which depends on the code instance, a function can't be used to capture the common parts of the code. The same goes if the parameter is a type. Usually, these two cases are the only times it is really necessary to use macros.

In fact, there are a few situations in which it is a serious mistake to use macros. In spite of "traditional wisdom," the use of macros for speed should be avoided. For one thing, modern RISC processors have very little procedure call overhead, so using macros in that case in hopes of gaining speed really doesn't buy you anything. Moreover, there is a tendency among programmers to be mistaken about which parts of a program have performance problems. When you don't really know where the performance bottlenecks are, creating macros for unnecessary optimizations is a waste of time and makes debugging harder. Finally, macros tend to increase the size of the executable program (not really the same problem as the size of the source code).

Because of limited size cache memories, unnecessarily repetitious code can actually make the program slower. In such situations, increased use of functions would actually be advantageous to cut down on repetition in the source code and in the executable.

5.5.2 Duplication of Information Between Code and Comments

One might think that commenting the code is always a good thing, but there is also a risk with commenting the code: namely, that information is duplicated. Comments that paraphrase code should be avoided. In contrast, comments that explain passages that might need additional explanation are generally useful, as well as comments that identify the author and explain the purpose of a module. Here are some kinds of comments that should be avoided.

5.5.2.1 Comments that Document Function Headers (a bad idea)

ANSI C function headers contain the name of the function the number of arguments, the name and type of each argument, and the type of the return value of the function. There

is no need for an additional comment explaining these things. Indeed, it is common that programs written where such additional comments are mandatory contain inconsistencies between the code and the comments. The explanation for such anomalies is simple: the maintainer is under pressure to get the code fixed and so modifies the program but not the comment.

5.5.2.2 Comments to Variable and Field Definitions (a bad idea)

The purpose of a variable or of a field in a structure should as much as possible be evident from their name. If additional information is necessary, a comment may be added, but then the comment should not repeat the name or the type of the variable, but instead give only the additional information. Rules that require a comment for each variable usually generate duplication of information and should thus be avoided.

5.5.2.3 Comments that Paraphrase Code (a bad idea)

Comments to pieces of code should provide additional information. Comments that only paraphrase code create maintenance nightmares. The classic example is:

```
i++; /* increment i */
```

but there are many others. Indeed, even comments that provide additional information can be avoided by replacing them with a function that contains the code to be commented. The name of the function provides the additional information. For instance, instead of explaining that certain lines invert a matrix, one could create a function called `invert_matrix` that contains the code to be commented. This is an example of an abstraction that is not created for multiple use, but only for the purpose of giving it a name.

5.6 A Word About Garbage Collection

Normally in C, we use the functions `malloc()` and `free()` to allocate and free memory. The problem is knowing when it is safe to free memory. Usually, the memory that has been allocated belongs to an object that has been created in some kind of module. In order to free the memory belonging to the object, we must be sure that no reference to the object still remains.

One way to guarantee that an object can be freed is to make sure that there is only one reference to it, or at least that the exact number of references is known at any instant. This method creates a problem when an object is inserted into a container, as the container itself keeps a reference. One solution to the problem would be to define, for each container, exactly who is responsible for freeing memory and when.

Unfortunately, this practice creates a problem related to maintainability. Here is an example:

Suppose we have a container that can insert and delete objects. When an object is deleted from the container, the container cannot free the memory that the object occupies, simply because the object may be a complicated, recursive data structure that needs to be freed in a particular order. Thus, client code must free this memory. However, if client code should free the memory, then we must clearly state that the container, after deletion of the

object, no longer keeps a reference to it. So far, so good. Yet, as we all know, software evolves, and a very reasonable compatible evolution of the container would be to add an undo facility, where objects that are deleted from the container would simply be saved on some kind of undo list so that they could be reinserted by a call to the `undo` function. But in the presense of an explicit rule stating that the container must not keep a reference to the object, this evolution is no longer upwardly compatible. Thus in order to make such an improvement, all existing client code must be examined and modified to take into account the additional reference. This situation is clearly undesirable.

One can thus argue that explicit memory management is an obstacle to maintainable code. The solution to the problem is to use automatic memory management, or a garbage collector. Such a garbage collector exists and it is called `cgc`. It also has the additional advantage of strongly encouraging modules based on data types rather than functions.

5.7 Idiomatic Expressions

Like natural languages, programming languages have their own idioms, or idiomatic expressions. As with natural languages, idioms identify the speaker as a native. Code that does not contain idioms where appropriate gives an impression of amateurish or childish programming practice. Using idioms from other languages creates an impression of foreigness. We don't want to give the impression, however, that using appropriate idioms is simply a question of taste or even snobbism; rather, the sound use of idioms means that the programmer participates in the shared culture of a given programming community and can thus benefit from programming traditions and conventions that make application code more legible and more easily maintained.

5.7.1 Array Traversal: a C Idiom

An example of a C idiom is the way an array is traversed. For example:

```
for(i = 0; i < n; i++)
   {
      a[i] = ...;
      ...
   }
```

Notice that the array is indexed from 0, that `<` is used rather than `<=`, that the loop counter is incremented with the `++` operator, and that the traversal is done by a `for` statement rather than a `while` statement, and that the loop counter is initialized in the first clause of the `for` statement. An entirely (unless the body contains the continue statement) equivalent statement, which is not a C idiom would be:

```
i = 1;
while(i <= n)
   {
      a[i - 1] = ...;
      i = i + 1;
   }
```

Since the two examples are semantically equivalent, you might well ask why we are insisting that the idiom is better. The idiom is the better choice because it represents shared knowledge and practice within the programming community, and as we indicated before, exploiting that shared experience makes code easier to read and maintain.

5.7.2 Foreign Idioms

Code that contains phrases that are not idioms or that are idioms from other languages is extremely hard to maintain and should be avoided.

While the example of array traversal is quite obvious even to a novice C programmer, the following may not be. The problem is to insert an integer into a sorted linked list of integers so that the list remains sorted. Here is the nonidiomatic C solution:

```
list l = (list) malloc(sizeof(struct list));
l -> val = val;
if(the_list == NULL || the_list -> val >= val)
   {
      /* insert the element first */
      l -> next = the_list;
      the_list = l;
   }
else
   {
      list q = the_list;
      list p = q -> next;
      for(; p && p -> val < val; q = p, p = p -> next)
        ;
      l -> next = p;
      q -> next = l;
   }
```

The nonidiomatic C solution uses a pointer `p` and a trailing pointer `q` so that we remember the place where the element is to be inserted. A special case is needed to treat the situation where the new element must be inserted first in the list. This situation arises either if the list is empty or if the first element is already greater than or equal to the new element to insert.

The idiomatic C solution uses the possibility for C to point not only to an object such as a structure or an array, but also to a field of a structure or to a component of an array.

```
list *p;
list l = (list) malloc(sizeof(struct list));
l -> val = val;
for(p = &the_list; *p && (*p) -> val < val; p = &((*p) -> next))
  ;
l -> next = *p;
*p = l;
```

We notice that there is no longer a special case. By using a pointer to a list (which itself is a pointer, by pointer abstraction), we can either point to the (supposedly global) variable `the_list`, or to the next field of any of the structures in the list. We thus eliminate the special case of insertion at the beginning of the list.

5.8 Useless Code

A situation to watch out for is one where useless code is introduced. Experienced programmers usually watch out for this, but it is a big problem in code written by beginners.

Useless code is usually the result of the programmer having identified a special case that in fact is not one. Most often, it happens in applications using pointers. Useless code tends

to confuse the reader causing him or her to spend considerable time trying to understand why a particular case is a special one.

It is hard to define exactly what useless code means, so instead we give an example that illustrates the essence of the most common case (but there is endless variation).

```
if(x -> next == NULL)
  {
     return NULL;
  }
else
  {
     return x -> next;
  }
```

Here, both cases yield the same result. Therefore, one can eliminate the test as well as the first block, giving simply:

```
return x -> next;
```

6 The C Header Files

Traditionally, C header files have been used to define structures and other data types to be shared by several programs or modules. Unfortunately, definitions of `structs` contain implementation details that should not be visible to client code.

For instance, let us suppose that we implement a graph where the nodes are defined as C structures. A function that traverses the graph may have to know which nodes have already been visited. One way of implementing such knowledge is by means of a flag field of the structure that represents the node. But, although this field should not be used by client code, there is no way to prevent the client from accessing it or modifying it if the field definition occurs in the structure declaration in the header file. Client code has to be recompiled whenever such implementation details are added or removed from the structure declaration. In the worst case, client code actually takes advantage of a field that may disappear in a future version of the module. Then client code may even have to be rewritten when implementation details of the module change.

The solution to the problem is to put in the header file only information that concerns the interface of the module, i.e., if the module is an implementation of an abstract data type, then the header file should contain only the operation on the data type.

Unfortunately, C makes it hard to produce such clean header files. The problem is that in order to define the operations in the form of C functions, the type of their arguments and return values must be specified. If either one is the data type to be defined by the module, then this type must be defined in the header file as well. Thus, we must accept to declare the implementation type of the abstract data type in the header file. But in line with our goal of hiding implementation details, we would give only the following two declarations:

```
struct datatype;
typedef struct datatype *datatype;
```

Unfortunately, we have to declare that `datatype` is a structure. Thus, in doing that, we have exposed an implementation detail. How serious is this breach of privacy? In all but the most trivial cases, we can usually say that this is not a severe problem. However, there are situations where this breach of privacy—this exposure of implementation details to the client—does cause difficulty. For example, it prevents us from changing between an implementation of a set as a structure (say) and as an integer that represents a bit-vector of fixed size, because, of course, the bit-vector is not a structure. In this face of this kind of difficulty, we have to console ourselves with the thought that most nontrivial data types will be implemented as structures anyway.

Fortunately, since the fields of the structure are not declared, we do not reveal any other implementation details. There is an apparent inconvenience for client code in this solution: that the size of the structure is unknown to client code, thus preventing client code from allocating objects of type `datatype`. In fact, this is not an inconvenience at all; it is a requirement. The size of the storage type that implements the data type is an implementation detail that should not be possible to discover from client code.

The rest of the header file contains ANSI C function prototypes that define the operations on the data type. In almost all cases, there will be no variable declarations in the C header file. For documentation purposes, the arguments given in the function headers should always have a name in addition to a type. The reason is that the type itself is not sufficient to indicate the purpose of the argument. A well chosen variable name makes a big

difference. A programmer writing a client module can use these names to verify that a call to the operation contains the right arguments in the right order. Operations that take no arguments should have an argument list containing the word `void`, so that the declaration will not be interpreted as a conventional header declaration with an unknown number of arguments of unknown types.

To avoid potential type clashes that may occur if the same header file gets included more than once in a program, directly or indirectly, we systematically put the following two lines at the beginning of the header file

```
#ifndef datatype_H
#define datatype_H
```

and at the very end we put the line

```
#endif
```

The header file should always be included in the `.c` file. This inclusion is the only way for the C compiler to verify that the interface actually corresponds to the implementation.

7 Cookbook Modularity

It is hard, if not impossible, to give a general-purpose method that is guaranteed to generate reusable and maintainable code. We nevertheless try here to give a step-by-step description of how to obtain good modular code. Such a description is necessarily very incomplete, but it can serve as a guideline during development.

7.1 Name an Abstract Data Type

The first step in the development of an application is to try to find out what objects or abstract data types the application is going to manipulate. Such objects often correspond to real-life objects, but are naturally only crude simplifications of their real-life counterparts.

Try to make a list of names of abstract data types that your application is going to use. Typical names of abstract data types are: `person`, `car`, `account`, `paycheck`, `engine`, `temperature`, `date`, but also purely imaginary objects of computer science such as `set`, `list`, `sequence`, `stack`, `queue`, `tree`, `hash table`, etc.

7.2 Define the Operations on the Abstract Data Type

The next step is to take each abstract data type and identify the operations that it will accept. For objects corresponding to real-life objects, this amounts to a simplification adequate for the application. For instance, if the application is to administer bank accounts, we would be interested in the name and phone number of the person, but not (say) his or her weight (even though those details may be useful in a medical application).

Not even the objects of computer science have clearly defined operations. Take `set`, for instance. One may think that there are standard set operations such as union intersection, etc. However, implementing a set is hard if generality and performance must both be obtained. Thus, if one can simplify the operations, even eliminate some, then a simpler and faster implementation may be obtained. For instance, if the only operations we really need on a set are these: (1) to add an element to the set, (2) to check whether the set is empty, and (3) to remove any object of the set, then a stack can be used to implement the set in a very efficient way.

In summary, your abstract data type is not completely defined until you have determined the operations that it is to support. In order to obtain closed modules, make sure that the set of operations is minimal and complete.

7.3 Decide the Kind of Interface: Functional or Imperative

The next step is, for each abstract data type, to decide whether its interface should be functional or imperative. A functional interface can be chosen when it is important to keep old versions of data structures around; for instance, in an application that needs to save a history of modifications. A functional interface is an easy way to insure that old versions of a data structure will be untouched by operations. A functional interface is also essential if referential transparency is necessary. A functional interface may require a garbage collector as operations may have to create large quantities of dynamically allocated heap memory.

On the other hand, if objects are heavily shared and if consistency between all the shared versions is desirable, then an imperative interface is more useful. Also, an imperative interface tends to use less memory, so one may be able to avoid using a garbage collector. (Notice, however, that a garbage collector is useful for other things as well.)

7.4 Write the Header File

Once you've decided whether to use a functional or imperative interface, and you've taken decisions about names into account, write the `.h` file with the operations in the form of ANSI headers and with function names prefixed with the name of the data type, or an abbreviation of the data type. Make sure the name of the header file reflects the name of the abstract data type. Try to avoid names such as `set`, since there may be many different implementations of sets with endless variation on the operations that are supported as well as their complexity.

Make sure that no implementation details are visible in the header file. If necessary, use pointer abstraction and polymorphism to make sure the operations do not need any compile-time parameters.

7.5 Decide the Complexity of the Operations

Once the exact definitions of the operations are known, you need to decide the time complexity of them. Unfortunately, this may be somewhat tricky, since choosing an algorithm with low complexity for one operation may make it necessary to use a more costly algorithm for another. The complexity is clearly going to depend on the implementation, so it is not possible to decide on complexity without having an idea of how the data structure is to be implemented.

In some sense, this is the first step towards implementation of the operations. Make sure the complexity of the operations are documented in the header file, unless, of course, they are obvious.

7.6 Implement the Operations

For each abstract data type, implement the operations in a `.c` file. The name of the `.c` file should have the same root as the corresponding header file. Make sure the header file is included in the `.c` file and that ANSI function headers are used so that the compiler can verify the type and number of the parameters.

The implementation must be faithful to the interface. Use a functional implementation for a functional interface and an imperative implementation for an imperative interface. Consider using object headers to respect this consistency.

Be sure to use algorithms that respect the decisions about the complexity of the different operations. Use C idioms as much as possible. Avoid foreign idioms or nonidiomatic solutions.

In the `.c` file, mark all variables and functions that are not in the header file with `static`. Avoid duplication of information, and use standard naming conventions. Use

widely accepted indentation. Comment where appropriate but consider rewriting the code that needs to be commented.

If dynamic memory allocation, but especially de-allocation looks like it is going to be a problem, consider using the `cgc` garbage collector.

8 Example: Ordered Set

As an example of the method presented in this document, we consider the creation of an ordered set, i.e., a container in which we can put elements that are totally ordered with respect to some ordering function supplied by the user. Ordinary set characteristics apply; for instance, that an element occurs only once in a set.

We suppose that some bottom-up method is applied to creating the application. It has been determined that the application needs to manipulate sets of elements. By a set, we mean a container that supports the following operations:

Creation of an empty set

Insertion of an element into a set

Test of membership of an element in a set

The union of two sets

In addition, we know that the data type of the elements contained in the set supports the equality test between two element, and furthermore, since the elements are totally ordered, we also know that there is a test to determine whether one element is smaller than another.

We suppose that the application needs to share ordered sets of elements and that modifications should be visible to all shared copies of the set. Therefore, we must have an imperative interface and implementation.

We have chosen to prefix the exported names of our data type with `oset_`. The header file will look like this

```
#ifndef OSET_H
#define OSET_H
struct oset;
typedef struct oset *oset;

typedef int (*lessfun)(void *x, void *y);

/* create a new empty ordered set. The argument is a function to compare
   two elements */
extern oset oset_create(lessfun less);
/* insert element only if it is not already a member */
extern void oset_insert(oset s, void *element);
/* return nonzero if element is member of set, otherwise zero */
extern int oset_member(oset s, void *element);
/* the first arg becomes the union of the two args */
extern void oset_union(oset s, oset t);
#endif
```

Notice that for the header file, we try to make comments that would be helpful to a programmer writing a client module.

```
#include "oset.h"

typedef struct list *list;

struct list
{
   void *element;
   list next;
};
```

```
struct oset
{
   lessfun less;
   list elements;
};

oset
oset_create(lessfun less)
{
   oset tmp = (oset) malloc(sizeof(struct oset));
   tmp -> less = less;
   tmp -> elements = 0;
   return tmp;
}

typedef list *position;

/* return the first position p greater than or equal to the one given such
   that either p is the last position (no element has that position) or
   the element given as argument is smaller than or equal to the one at p */

static position
posit(position initial_pos, void *element, lessfun less)
{
   position p = initial_pos;
   for(; *p && less((*p) -> element, element); p = &((*p) -> next))
     ;
   return p;
}

/* insert an element at the position given if and only if the element at
   that position is not equal to the one we want to insert */

static void
insert_if_nonequal(position p, void *element, lessfun less)
{
   if(less(element, (*p) -> element))
     {
        /* then the elements are not the same, insert the new one */
        list l = (list) malloc(sizeof(struct list));
        l -> next = *p;
        l -> element = element;
        *p = l;
     }
}

void
oset_insert(oset s, void *element)
{
   position p = posit(&(s -> elements), element, s -> less);
   insert_if_nonequal(p, element, s -> less);
}

int
oset_member(oset s, void *element)
{
   list *lp = posit(&(s -> elements), element, s -> less);
```

```
   return !(s -> less(element, (*lp) -> element))
}

void
oset_union(oset s, oset t)
{
   position p = &(s -> elements);
   list l = t -> elements
   for(; l; l = l -> next)
     {
        p = posit(p, l -> element, s -> less);
        insert_if_nonequal(p, l -> element, s -> less);
     }
}
```

9 Applications to other languages

9.1 C++

Since C++ is in some sense a superset of C, the methods described above may be used in C++ as well. However, since class definitions would have to be in the `.c` file according to our rules, we are unable to use inheritance. If we want to use inheritance, we must put the class definitions in the `.h` files, but in doing that, we violate the principle that client code should not have to be compiled as a result of modifications to a module.

Furthermore, if we put the class definitions in the `.c` file as this guide suggests, we cannot define the operations as methods of a class. Again, if we want the operations to be methods of the class, we must put the class definition in the `.h` file.

One solution would be to put the public part of the class definition, i.e., the operations, in the `.h` file and then derive a class in the `.c` file that contains all the implementation details. This is probably the solution that comes closest in spirit to the C solution. Unfortunately, we are now unable to use inheritance of implementations since all implementation is in the `.c` file.

9.2 Pascal

Standard Pascal does not have modules, so it is very hard to write maintainable and reusable code in standard Pascal. But even versions with modules may have problems. Anonymous records would be necessary in order to hide implementation details to client code. Some kind of generic pointer type would be necessary. Pointers to functions are supported in Pascal only for arguments to other functions. Thus we could not save a pointer to a function in a record for later application. We would have to give the comparison function as an argument to all operations that need to use it. Finally, Pascal does not have pointers to elements of records, except as reference paramenters to functions. With a bit of juggling, one can usually live with such a solution.

10 Documenting Your Code

Documentation is an essential part of a program. One can distinguish between three major categories of documentation: project documentation, maintenance documentation, and user documentation. As we consider these categories of documentation, we need to set up a protocol about which documents refer to which others so that we do not inadvertantly create loops of cross-references. We do not want to create a situation where the reference manual says, "See the user's manual about this topic" but the user's manual says, "See the reference manual." To that end, we'll say that project documentation can refer to documents in the other categories and maintenance documents can refer to user documentation, but not to project documentation. User documentation must be autonomous, since, if you think about it, the end-user probably does not have access to project nor maintenance documentation.

10.1 Project Documentation

Project documentation contains all information not directly related to the program itself. It may contain comparisons to other similar products, alternative solutions to programming problems, projects schedules, cost, staff, etc. In a programming project, each phase of the development may be concluded with some kind of project documentation.

10.2 Maintenance Documentation

There are two essential documents considered as maintenance documentation. The first and most important one is the code itself. It is in some sense the only true maintenance document, as it formally specifies all aspects of what the program does. It is therefore extremely important that the code be written to be understandable to other programmers. Of course, there is no point in trying to make the code understandable to all programmers. You may assume that the programmer that is to read the code is a "native speaker" of the programming language that is used, i.e., that he or she knows the idiomatic expressions of the language.

It is an error to think that the code is only meant for the compiler. The fact that the compiler can directly understand the code should be considered a lucky coincidence. The code should be written mainly for other people to read.

The other important document in this category is the maintenance manual, which contains information that does not belong in any particular place in the code. A typical example is architectural considerations that concern the way the program was divided into modules, and other globally visible decisions. Low level information, such as algorithms used for a particular operation, are best documented as comments in the code itself. Under no circumstances does the maintenance manual duplicate information from the code. In particular, it does not contain a complete list of all the functions in the code, for which there are tools such as `etags`. Nor does it document trivial data structures (such as lists or binary trees), or trivial algorithms (such as linear search or quicksort).

Well-known algorithms and data structures can be documented by giving the corresponding structure or function an appropriate name. You may assume that the maintainer

is familiar with basic data structures and algorithms. More esoteric or more recent, but still widely known, data structures and algorithms can be documented by giving a reference to the corresponding publication in a comment in the code. Only algorithms and data structures specific to the applications should be described completely. If the description is short, put it in the comment in the code. If the description is complicated or requires theoretical analyses, then the maintenance manual may be appropriate for this information. In the latter case, make a reference in the code to the chapter in the maintenance manual discussing the data structure or the algorithm.

10.3 User Documentation

The final category is the one containing user documentation. In this category we find the reference manual, the user manual, quick reference cards, and tutorials.

The reference manual contains everything there is to know about using the application. The main use of the reference manual is by experienced users that need to find some detail that they might have forgotten or simply never knew existed.

Therefore, it is important for the reference to contain a good index and a complete table of contents. Indexing a manual is hard, so care should be taken. If possible, consult a professional technical writer about indexing.

The reference manual is not usually read from beginning to end. In fact, you can never be sure about the order in which a reader encounters the topics in a reference manual. While you should give some thought to a logical order for presenting these topics in the reference manual, you should also make it possible for a reader to access topics randomly and still learn all he or she needs to know. For that reason, including cross references within the reference manual becomes extremely important.

To prevent frustrating loops within the documentation as a set, the reference manual cannot refer to any other documentation, including the user manual and tutorials. The reference manual should be the "last word" on every topic of interest to the user.

The user manual is often task-oriented. It explains, for each task that the user may need to accomplish, a sequence of manipulations that accomplish that task. The user manual is not necessarily complete. It may handle only the most common cases. References to the reference manual are acceptable for cases that are not explicitly discussed. References to the reference manual are also helpful just to point the user to the correct place in the ultimate document. However, under no circumstances should the user manual refer to a tutorial.

The user manual is often read from beginning to end (at least the first time). Therefore, the user manual can define concepts that are necessary for the rest of the manual in the first chapter. Such definitions are generally a good idea, but make sure that the concepts are used in a consistent way in all user documents. Since the user manual may be read out of sequence, be sure to refer to definitions and concepts whenever their use requires complete understanding.

A tutorial is even lighter than the user manual. It usually shows only a small part of what the program can do, but it does so in a step-by-step way, often supposing that the user is manipulating the program simultaneously while reading the tutorial. A tutorial can refer to both the user manual and the reference manual.

Finally, a reference card can be seen as an extremely short version of the reference manual. In some cases, it may be necessary, however, to leave out certain material that requires considerable space.

Concept Index

A

abstract data type 4, 26
abstraction, functional 19
abstraction, macro- 19
abstraction, pointer- 10, 27
algorithm, complexity of 7, 27
ANSI function prototype 24, 27
application programming 1
arbitrary limitation 7
`assert` clause 12
automatic memory management 2

B

bottom-up design 4, 29
bug fixes, time spent on 3

C

C language 2
C++ programming language 32
cache memory, size of 19
cast, type- 10
`cgc` garbage collector 6, 10, 16, 21, 28
closed module 15
code as documentation 33
code, maintainable 1
code, reusable 1
code, useless 22
comment 28, 29
comment, may duplicate information 19
complete interface 15
complexity of algorithm 7, 27
computer science objects 1
container module 9, 10, 29
container, polymorphic 10
contract, programming by 11
cut-and-paste, source of duplication 19

D

data type, abstract 4, 26
data-driven programming 10
defensive programming 11
design, bottom up 4, 29
design, top-down 4
documentation 33
documentation, maintenance- 33
documentation, project- 33
documentation, user- 33, 34
duplication, of information 18, 27

E

encapsulation 10
encapsulation of information 2
equality, preservation of 6
error 12
error message 13
error, fatal 12
error, indicated by function return code 14
event loop 10
evolution of a program 3
exception 13
exception, handled by `setjmp` and `longjmp` 14
expression, idiomatic 21

F

file descriptor 16
file pointer 16
FORTRAN 14
function prototype 24, 27
function return code, to indicate error 14
function, module as 3, 4
functional abstraction 19
functional implementation 6
functional interface 6, 26

G

garbage collection 2, 10, 20
garbage collector 10, 20, 26
generic module 9
genericity 9
genericity, in C 9
`goto` instruction, nonlocal 14

H

hash table, alternative to `switch` 11
header file 2, 5, 10, 24, 27
header of an object 6
heap memory 10, 20, 28
hiding implementation details . . 2, 4, 10, 15, 24, 27
Hungarian naming convention 15

I

identifier, length of 16
idiom 21
idiom, foreign 22
idiom, of C 22, 27
idiomatic expression 21
imperative implementation 6, 29
imperative interface 6, 26, 29

implementation details, hiding . . 2, 4, 10, 15, 24, 27
implementation of a module 5
implementation, functional 6
implementation, imperative 6, 29
indentation 28
indentation, consistent 17
information, duplication of 18, 27
inheritance 32
instantiation of type variable 9
interface of a module 5
interface, complete 15
interface, functional 6, 26
interface, imperative 6, 26, 29

K

keyword, `static` 2, 17, 27

L

language, C 2
length of identifier 16
library, shared 1, 8
limitation, arbitrary 7
limitation, soft 7
`longjmp` library routine 14
loop counter 17

M

macro abstraction 19
maintainable code 1
maintenance documentation 33
maintenance manual 33
manual, maintenance- 33
manual, reference- 34
manual, user- 34
memory management, automatic 2
memory, dynamically allocated 10, 20, 28
memory, heap- 10, 20, 28
modular programming 3
module, closed 15
module, container- 9, 10, 29
module, functional 3, 4
module, generic 9
module, object- 3, 4
module, open 15
module, polymorphic 9, 10
module, precompilation of 8
module, type-safe. 10

N

naming convention 15, 27
naming convention, Hungarian 15
naming convention, traditional 16
nonlocal `goto` instruction 14
numeric problems 3

O

object, header of 6
object, module as 3, 4
object, shared 2, 29
objects, of computer science 1
objects, real world 1
open module 15
open-closed principle 15
operation, on an abstract data type 26

P

parameter of software component 8
parameter, compile-time 27
parameter, compile-time- 8
parameters of software component 3
pointer abstraction 10, 27
polymorphic container 10
polymorphic module 9, 10
polymorphic module, size of object in 9
polymorphism 9, 27
precompilation of module 8
prefixing fields of a structure 16
prefixing names of a module 16, 27, 29
preserving equality 6
program execution, termination of 12
program, evolution of a 3
programming, application- 1
programming, by contract 11
programming, data-driven 10
programming, defensive 11
programming, modular 3
programming, structured 3
programming, symbolic 1, 3
programming, system- 1
project documentation 33

Q

qsort function 10

R

real world objects 1
reference card 34
reference manual 34
reference, to an object 20
referential transparency 26
reusability 3
reusable code 1

S

`setjmp` library routine 14
shared library 1, 8
shared object 2, 29
size of object in polymorphic module 9
soft limitation 7
software component 3
`static` keyword 2, 17, 27
structured programming 3
`switch` statment 11
symbolic programming 1, 3
system programming 1

T

termination of program execution 12
top-down design 4
Turbo Pascal 14
tutorial 34
type cast 10
type safe module 10
type variable 9
type variable, instantiation of 9

U

useless code 22
user documentation 33, 34
user manual 34

V

variable, type- 9
visibility of names 17

Short Contents

1 Introduction 1
2 The C Language 2
3 Reusability and Maintainability 3
4 How to Write Reusable Code 4
5 How to Write Maintainable Code 15
6 The C Header Files 24
7 Cookbook Modularity 26
8 Example: Ordered Set 29
9 Applications to other languages 32
10 Documenting Your Code 33
Concept Index 36

Table of Contents

1 Introduction 1

2 The C Language 2

3 Reusability and Maintainability 3

3.1 Reusability 3
3.2 Maintainability 3

4 How to Write Reusable Code 4

4.1 Functional Modules or Data Modules 4
4.1.1 Functional Modules 4
4.1.2 Data Modules 5
4.2 Interface vs. Implementation 5
4.3 Functional vs. Imperative Interface and Implementation 6
4.4 Absence of Arbitrary Limitations 7
4.5 Few Assumptions 7
4.6 Few Parameters 8
4.7 Precompilation of Module 8
4.8 Genericity vs. Polymorphism 8
4.8.1 Genericity 9
4.8.2 Polymorphism 9
4.9 Pointer Abstraction 10
4.10 Data-driven Programming 10
4.11 Programming by Contract 11
4.12 Error Situations 12
4.12.1 Fatal Errors 12
4.12.1.1 Contract Breaches 12
4.12.1.2 Contractual Fatal Errors 13
4.12.2 Nonfatal Errors 13
4.13 Exceptions 13

5 How to Write Maintainable Code 15

5.1 The Open-Closed Principle 15
5.2 Naming Conventions 15
5.2.1 Hungarian Naming Conventions (a bad idea) 15
5.2.2 Prefixing Fields by the Structure Name (a bad idea) 15
5.2.3 Module Prefix 16
5.2.4 Traditional Naming 16
5.2.5 Short Names and Long Names 16
5.3 Indentation 17
5.4 Visibility and Scope 17

5.5 Normalization . . . 18
5.5.1 Duplication of Code . . . 18
5.5.2 Duplication of Information Between Code and Comments . . . 19
5.5.2.1 Comments that Document Function Headers (a bad idea) . . . 19
5.5.2.2 Comments to Variable and Field Definitions (a bad idea) . . . 20
5.5.2.3 Comments that Paraphrase Code (a bad idea) . . . 20
5.6 A Word About Garbage Collection . . . 20
5.7 Idiomatic Expressions . . . 21
5.7.1 Array Traversal: a C Idiom . . . 21
5.7.2 Foreign Idioms . . . 21
5.8 Useless Code . . . 22

6 The C Header Files . . . 24

7 Cookbook Modularity . . . 26

7.1 Name an Abstract Data Type . . . 26
7.2 Define the Operations on the Abstract Data Type . . . 26
7.3 Decide the Kind of Interface: Functional or Imperative . . . 26
7.4 Write the Header File . . . 27
7.5 Decide the Complexity of the Operations . . . 27
7.6 Implement the Operations . . . 27

8 Example: Ordered Set . . . 29

9 Applications to other languages . . . 32

9.1 C++ . . . 32
9.2 Pascal . . . 32

10 Documenting Your Code . . . 33

10.1 Project Documentation . . . 33
10.2 Maintenance Documentation . . . 33
10.3 User Documentation . . . 34

Concept Index . . . 36