

C++ Programming Techniques

2nd Edition

Practical C++ Programming



O'REILLY®

Steve Oualline

SECOND EDITION

Practical C++ Programming

Steve Oualline

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Paris • Sebastopol • Taipei • Tokyo

Program Design

If carpenters made houses the way programmers design programs, the first woodpecker to come along would destroy all of civilization.

—Traditional computer proverb

Now that you've got the mechanics of programming down, we'll take a look at how to design good code. Creating a well-designed program is both a science and an art form. There is a lot of science involved in the techniques and rules used to produce a good design. Your artistic side comes in to play when you use these rules to lay out a well-designed and beautiful program.

Design Goals

Before we decide how to make a well-designed program, we need to define what we mean by well designed. Different people value different things. But in most cases, people tend to value the same attributes. So let's explore what people value in a program:

Reliability

People want a program that works. Crashes are extremely frustrating. They cost people time, cause data loss, and in extreme cases can cost people's lives. So reliability is extremely important.

Economy

Most people, especially managers, don't like to spend money. They want the cheapest software possible.

Ease of use

No program is useful if people can't use it. This may sound a bit obvious, but lots of programmers suffer from the "added feature disease" where they want to cram as many features as possible into their code. The result is something overly complex and difficult to use: in other words, a badly designed program.

Design Factors

Now that we know what we want in a design, let's see what we need to get there. As the science of programming has developed, people have discovered several factors that go into a good program design. You don't have to create a program with these factors in mind, but most programs that do a good job of satisfying the design goals listed above are designed with these factors in mind:

Simplicity

The simplest code that does the job is usually the best. It's the most reliable because it has the fewest things that can go wrong. Doing as little as possible also means that the code is cost-efficient to produce.

There are other benefits as well. Simple code is easy to maintain and enhance, so future programming costs are reduced, too.

Finally, the less you give a user to do, the less he can screw up. That generally means that the simplest software is the easiest to use.

Information hiding

A good design provides a minimal interface to the user and hides as many of the details as possible.

Expandability

Can the program be quickly and easily expanded? For example, suppose you are writing a word processor. How difficult would it be to use a different character set, such as Spanish or Korean? Better yet, how difficult would it be to adapt it to a more complex character set, such as Hebrew or Chinese?

Remember when PCs first came out: the maximum amount of memory you could have was 640K, and the early MS-DOS filesystem (FAT12) couldn't handle a disk bigger than 16MB. Both these limits have proven to be terribly inadequate.

Testable

If your program can be tested and tested easily, it will be tested. Testing leads to reliability. (Note that extensive testing should never be used as a substitute for a good design. Testing can only show the presence of defects, not their absence.)

But programs have a life of their own. They don't stop with version 1.0. There are a few design attributes that are appreciated by the people who have to maintain and enhance your code.

Reusability/generality

Writing code takes a lot of time and effort. Using already written code (assuming it does the job) takes very little time and effort. Good code design calls for your code to be reusable whenever possible.

For example, the linked list structure is extremely useful. In C everyone tends to create her own linked list package (a linked list of names, a list of messages, a list of events, etc.). The result is a lot of redundant code.

C++ has the STL, which contains a container class (`std::list`) that gives you all the features of a linked list. (Most of the time it's implemented as a linked list, but you don't have to know about the implementation details.) Because this container is generic, C++ programmers don't have to keep re-creating the same code over and over again.

A good design makes maximum use of reusable code and at the same time is itself designed so that it can be reused.

Different circumstances require different design criteria. I know of one oil company that spends lots and lots of money on making sure that their software is correct and extremely accurate. This software has one job: simulating how one pipe threads on to another. I asked them why they had a whole department devoted to pipe threads and they told me, "When a thread fails, we generally lose about fifty million dollars and a dozen lives."

Their design values accuracy and verifiability. They want to be really sure that their numbers are right. Lives depend on them.

On the other hand, I wrote a program to replace the standard Unix command *vacation* because the existing Unix command wouldn't work with our security system. The command was released with minimal testing and other reliability checks. After all, the existing command didn't work, and if the new command didn't work, the users didn't really lose anything.

Design Principles

There are a couple of basic design principles you should keep in mind when creating your design. They will help you create a design that not only works, but is robust and elegant.

The first is "Think, then code." Far too many people, when given an assignment, can't wait to start coding. But the good programmers spend some time understanding the problem and studying all aspects of it before they start coding. After all, if you are driving from San Diego to Chicago, do you jump in the car and head north-east, hoping you'll get there, or do you get out a map and plan your route? It's a lot less trouble if you plan things before you start doing.

The other design principle is "be lazy" (a.k.a. efficient). The easiest code you'll ever have to implement and debug is the code that you designed out of existence. The less you do, the less that can go wrong. You'll also find that your programs are much simpler and more reliable.

Design Guideline: Think about a problem before you try to solve it.

Following Orders

One of my favorite movies is called *The King of Hearts*. It is set in World War I. My favorite scene is where the general is giving three commandos their orders. This trio is a crack unit, well-known for instantly following orders.

“Men,” begins the General, “I want you to leave right away.”

The three turn around and run off.

“Stop” shouts the General. They halt and turn around. “Where do you think you’re going?”

“No idea, sir,” they answer in unison.

Far too many programmers act like these commandos. They run off and start typing on the keyboard before they have an idea where they are going.

Design Guideline: Be as efficient and economical as possible.

Coding

We’re going to start our discussion at the bottom and work our way up. The smallest unit of code that we design is the procedure. Procedures are then used to build up more complex units, such as modules and objects. By starting simple and making sure our foundation is good, we can easily add on to create more complex, yet robust programs.

Procedure Design

A procedure in C++ is like a paragraph in a book. It is used to express a single, coherent thought. Just as a paragraph deals with a single subject, a procedure should perform a single operation. Ideally you should be able to express what a procedure does in a single simple sentence. For example:

This procedure takes a number and returns its square.

A badly designed procedure tries to do multiple jobs. For example:

Depending on what values are passed in, this function will

1. allocate a new block of memory on the heap,
2. delete a block of memory from the heap, or
3. change the size of a heap block.

It’s the body of the procedure that does the actual work. A procedure should do its job simply and coherently. In general, programmers design and work on an entire procedure at a time, so the procedure should be small enough that the whole proce-

cedure can fit in a programmer's brain at one time. In practice this means that a procedure should be only one or two pages long, three at the most.

Design Guideline: Procedures should be no more than two or three pages long.

Procedure interface

The public part of a procedure is its prototype. The prototype defines all the information needed by the compiler to generate code that calls the procedure. With proper commenting and documentation, the prototype also tells the programmer using the procedure everything he needs to know. In other words, the prototype defines everything that goes into and out of the procedure.

Global variables

All the variables used by a procedure are either local to the procedure or parameters *except* for global variables. (The word “except” is an extremely nasty word. Frequently it indicates a complication or extra rule. Things were probably simple before the “except” came into the picture.)

The use of a single global variable inside a procedure makes the whole procedure much more complex. For example, suppose you want to know what a procedure does for a given call. If that procedure uses no global variables, all you have to do is look at the parameters to that procedure to figure out what is going to happen.

You can determine what the parameters are by looking at a single line in the caller. All the other variables are local to the procedure. That's only three pages long, so you probably can figure out what happens to them.

But now let's throw in a global variable. That means that the input to the procedure is not only the parameters, but the global variable. So who sets it? Because the variable is global, it can be set from just about anywhere in your program. Thus, to determine the input to a procedure, you must analyze not only the caller, but also all the code in the entire program. I've seen people do string searches through tens of thousands of files trying to find out who's setting a global variable.

Figure 26-1 shows the information flow into and out of a procedure and how this is affected by global variables

One way people try to get around this problem is to require that all programmers list the global variables used by their procedures in the heading comments to the function. There are a couple of problems with this. First of all, 99.9% of the programmers don't do it and the other 0.1% don't keep the list up to date, so it's totally useless. In addition, knowing that a procedure uses a global variable doesn't solve the problems caused by not knowing when and how it is used by the outside code.

Design Guideline: Use global variables as little as possible.

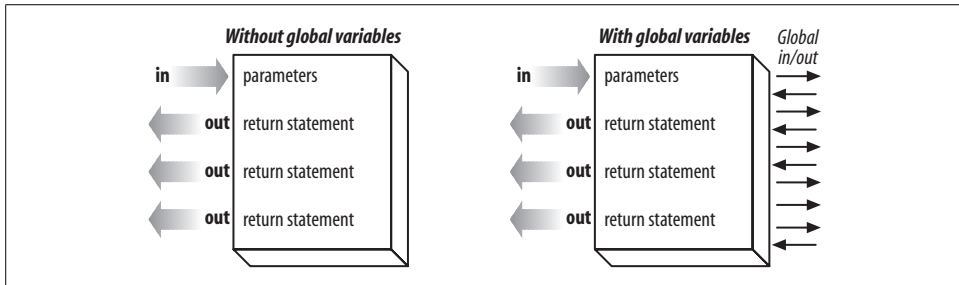


Figure 26-1. Procedure inputs and outputs

Information hiding

A well-designed procedure makes good use of a key principle of good design: information hiding. All the user of a procedure should see is the prototype for the procedure and some documentation explaining what it does. The rest is hidden from him. He doesn't need to know the details of how the procedure does its job. All he needs to know is what the procedure does and how to call it. The rest is irrelevant detail, and hiding irrelevant details is the key to proper information hiding.

Or as one of my clients said, "Tell me what I have to know and shut up about the other stuff."

Coding details

There are some coding rules for procedures that have been developed over time; if used consistently, they make things easier and more reliable:

1. For every C++ program file (e.g., *the_code.cpp*), there should be a corresponding header file (e.g., *the_code.h*) containing the prototypes for all the public procedures in the C++ file. This header file should contain only the procedures for the corresponding C++ file. Don't put functions from multiple program files in a single header file.
2. The C++ program file and the header file should have the same name with different extensions, for example, *the_code.cpp* and *the_code.h*.
3. The C++ program file should include its own header file. This lets the C++ compiler check to make sure that the function prototype is consistent with the function implementation.

Modules and Structured Programming

A collection of closely related procedures in a single file is called a *module*. Modules are put together to form a program. The proper organization of modules is a key aspect of program design.

First, your module organization should be as simple as possible. Figure 26-1A shows a program with seven modules. With no organization, there are 42 connections between the modules.

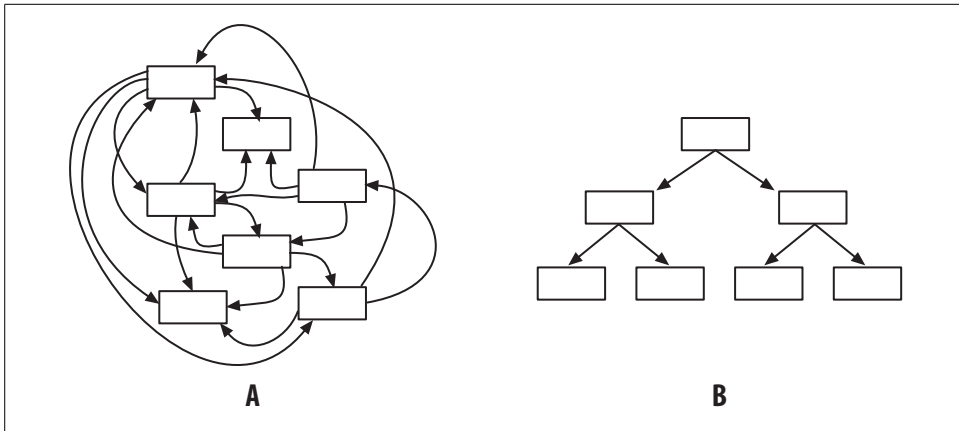


Figure 26-2. Module interactions

A programmer who is debugging a module must make sure that the other six modules he deals with work. Any problems in them are her problem. Testing such a system is a problem as well. To test one module, you need to bring in the other six. Unit testing of a single module is not possible.

Now consider the organization in Figure 26-1B. This system uses a hierarchical module organization. Consider the benefits of this organization. The modules at the bottom level call no one, so they can be tested in isolation. After these modules pass their unit tests, they can be used by the other modules.

People working on the middle-level modules have to contend with only two sub-modules to make sure their module works. They have some assurance their modules work—after all, they did pass the unit test—so the middle-level programmers can concentrate on dealing with their own modules.

The same thing holds true for the person dealing with the top-level modules.

By organizing things into a hierarchical structure, we've added order to the program and limited problems.

Design Guideline: Arrange modules into an organized structure whenever possible.

Interconnections

Although Figure 26-2 indicates that one module calls another, it doesn't show the number of calls that are being made. If we've done a good job hiding information, that number is minimal.

Let's first take a look at an example of what not to do to. We have a module that writes data to a file. Some of the procedures are:

```
store_char -- Stores a character in the buffer
n_char -- Returns the number of characters in the buffer
flush_buffer -- Writes the buffer out to disk
```

When we want to write a character to the file, all we have to do is put the character in the buffer, check to see if the buffer is full, and, if it is, flush it to disk. The code looks something like this:

```
store_char(io_ptr, ch);
if (n_char(io_ptr) >= MAX_BUFFER)*
    flush_buffer(io_ptr);
```

This is an extremely bad design for a number of reasons. First, to write a single character to a file, the calling function must interact with the I/O module four times. Four? There are only three procedure calls. The fourth interaction is the constant `MAX_BUFFER`. So we have four connections where one would do.

One of the biggest problems with the code is the poor effort at information hiding. For this program, what does the caller need to know to use the I/O package?

- The caller must know that the I/O module is buffered.
- The caller must know the sequence of functions to call to send out a single character.
- The caller knows that the I/O package uses fixed size buffers. (The fact that `MAX_BUFFER` is a constant tells us that.)

All of this is information the caller should not need to know. Let's look at an alternative interface:

```
write_char(io_ptr, ch) -- Sends a character to a file.
```

This function may buffer the character, but it may not. All the caller needs to know is that it works. How it works is irrelevant. In other words, the system may be buffered, unbuffered, or use a hardware assist. We don't know and we don't care. The character gets to the file. That's all we care about.

Back to our original three-function call interface. Let's see what problems can occur with it. First, the caller must call the proper functions in the proper sequence each time. This is a needless duplication of code.

There is also a maintainability problem. Suppose we decided that fixed-size buffers are bad and wish to use dynamic buffers. We'll add a function call `get_max_buffer` to our module. But what about all the modules out there that have `MAX_BUFFER` hard-

* The greater than or equal comparison (`>=`) is used instead of equal (`==`), as a bit of defensive programming. If somehow we overflow the buffer (`n_char(io_ptr) > MAX_BUFFER`), we'll flush the buffer and the program will continue safely.

coded in them? Those will have to be changed. Because we have used poor information-hiding techniques, we have created a maintenance nightmare for ourselves.

One final note: a better design would encapsulate the `io_ptr` data structure and all the functions that manipulate it in a single C++ class, as we will see later on in this chapter.

Real-Life Module Organization

Let's see how a set of modules can be organized in real life. In this case we are dealing with a computer-controlled cutter designed to cut out tennis shoes. The major components of this device are:

- A computer that controls the machine. This computer is also used to store the patterns for the various shoes.
- A positioning device for the cutting head.
- An operator control panel (lots of buttons and indicators).

The basic design results in five major modules:

1. The workflow module. This module is responsible for scheduling the various cutting jobs that come up (e.g., do ten batches for size 9, then twelve of size 11, and so on).
2. The positioning control system, which is responsible for moving the cutting head around and doing the actual cutting.
3. A hardware-monitoring system. Its job is to check all the status indicators and make sure that all the equipment is functioning correctly. (There's a lot of little stuff, such as oil filters, blowers, air filters, air supply, and so on, that all needs to work or we can't cut.)
4. The control panel input module. This module is responsible for handling any buttons that the operator pushes.
5. The control panel output module. All the blinking lights are run from this module.

A diagram of the major pieces can be seen in Figure 26-3.

This organization, although not quite hierarchical, is quite simple. Each module has a well-defined job to do. The modules provide a small, simple interface to the other modules.

The other thing about the modules is that they are designed to be independently tested. For example, when the project started, the machine didn't exist. What we had was a computer, a pile of parts, and a lot of stuff on back order. Since the workflow manager didn't require any hardware, it was developed first. The other modules were faked with test routines. The fake routines used the same interface (header files) as the real ones. They just didn't do any real work.

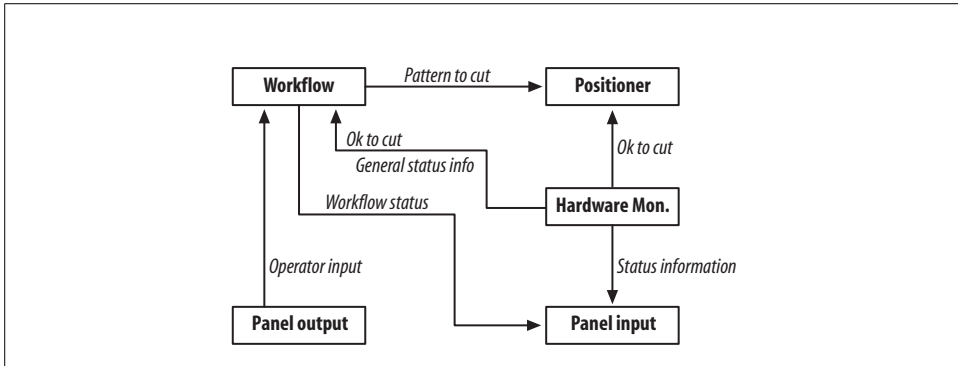


Figure 26-3. Cutting system, module design

It is interesting to note that the unit tests were used to test not only the software but the hardware. The unit test for the positioner module consisted of a front-end that sent various goto commands to the system. The first few tests were a little hairy because the limit switches had not been installed on the hardware, and there was nothing to prevent us from running the cutting head past the end and damaging the carriage. (Actually, the limits were rigorously enforced by a nervous mechanical engineer who held his hand inches above the emergency power off for the entire test. He wasn't about to let our software damage his hard work.)

This module structure let us create something that was not only simple, but testable. The result is increased reliability and decreased integration and maintenance costs.

Module Summary

So far we've learned a lot about how to design and organize our code. But programming deals with data as well as code. In the next few sections, we'll see how to include data in our design through the use of object-oriented design techniques.

Objects

So far our discussion has been focused on procedures. That's because procedures are one of the basic building blocks of a program. The other major piece of the puzzle is data structures. Combine the two and you have an object.

The design guidelines for a simple object are much like the ones for a module. You want to create a simple interface, hide as much information as possible, and keep the interconnects between objects to a minimum.

But objects give us one big advantage over the simple data structure/module design. With simple modules, your view of the data is limited. You either see the whole structure or you don't. Thus it's not possible (without getting very tricky) to write a

Testing “Right”

The Camsco Waterjet cutter was the first machine ever designed and built to cut out tennis-shoe insoles with a high-pressure jet of water. Because it was a cutting-edge design, a lot of tuning and adjustment was required. In fact, the system was run for about a year before it was shipped to the customer.

During that testing phase, we had an agreement with the shoemaker: they would supply us with free material for testing if we gave them the cut parts.

To get proper performance data, we always used the same size in almost all of our tests, 9-right. After each test we boxed up the cut parts and shipped them to the shoemaker so they could make shoes out of them. Or so we thought.

Just as we were boxing up the machine for shipment, we got a call from the plant. “Are you the people who keep sending the 9-rights to us each month?”

We confirmed that we were.

“Well, I’m glad I finally tracked you down. Purchasing had no idea who you people are, so you were a little hard to find. Do you realize that you’ve shipped us ten thousand 9-rights and no lefts?”

Evidently the first production run for our machine was ten thousand 9-lefts.

procedure that accesses the common elements of several different types of data structures.

Perhaps an example will explain this better. On the weekends, I’m a real engineer at the Poway-Midland Railroad. There are three major types of locomotives: steam, diesel, and electric. They have some attributes in common; they all pull trains, for example. But there are some attributes unique to each locomotive. For example, only a steam engine requires lots of water to operate.

Using simple data structures, it’s impossible to design a single data structure that encompasses all three locomotive types without wasting space. For example, the following structure describes all three types of locomotives—not well, but it does describe them:*

```
struct locomotive {
    bool running; // Is the locomotive running
    int speed; // Speed in MPH
    int num_cars; // Number of cars it can pull

    int water; // Water consumption in gallons / hour
                // [Steam engine only]
```

* I know that there are steam locomotives that burn things other than coal, but for the purposes of this example, I’m simplifying the universe and steam engines burn only coal.

```

    int coal;          // Coal used in tons / hour
                      // [Steam only]
    int diesel_oil;   // Oil consumed in gallons / hour
                      // [Diesel only]

    // .. rest of the data
};

```

Arranging data in this way is neither simple nor efficient. Objects let you arrange data in a new way by letting you create a general base object and derive more complex objects from it.

For example:

```

class generic_locomotive {
public:
    bool running;      // Is the locomotive running
    int speed;         // Speed in MPH
    int num_cars;      // Number of cars it can pull

    // ... rest of the data
};

class steam_engine: public generic_locomotive {
public:
    int water;        // Water consumption in gallons / hour
                      // [Steam engine only]
    int coal;         // Coal used in tons / hour
                      // [Steam only]
};

class diesel_locomotive: public generic_locomotive {
public:
    int diesel_oil;   // Oil consumed in gallons / hour
                      // [Diesel only]

    // .. rest of the data
};

class electric_motor : public generic_locomotive {
public:
    // Electric Locomotives aren't that complex

    // .. rest of the data
};

```

This data organization gives us tremendous flexibility. We are no longer constrained to writing procedures that deal with data structures as whole. Instead, procedures that want to deal with a generic locomotive can deal with the data type class locomotive. Other procedures that are locomotive-type-dependent can deal with their type of locomotive.

So our procedures can deal with the data at different levels. Thus with derived objects we've created different views into our data.

This is a good example of information hiding. Functions that deal with generic locomotives don't have to know about the specifics of each engine. They deal only with the generics. Figure 26-4 shows this information layering technique.

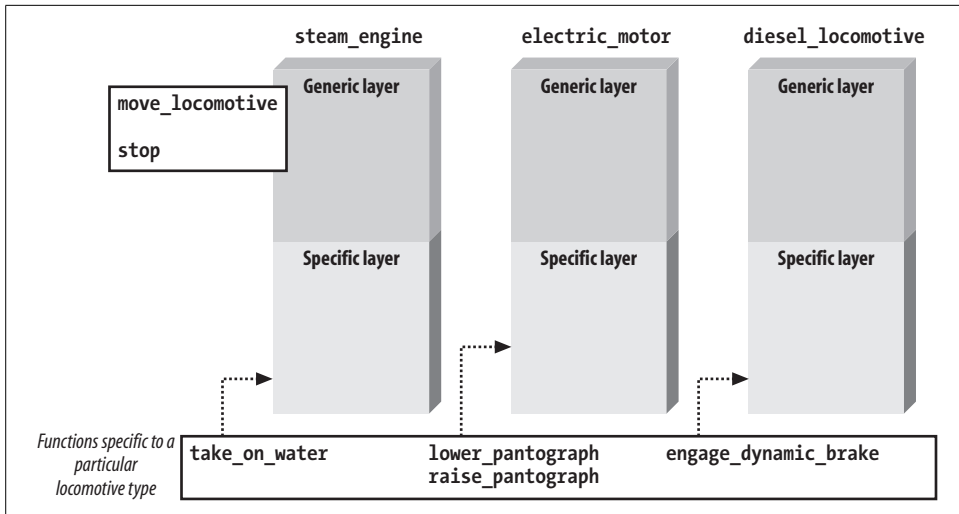


Figure 26-4. Layers of information

One of the nice things about virtual member functions is that they aid in information hiding. They provide an interface that is seen in the base class, but whose specifics reside in the derived class.

Interfaces and C++ Classes

Ideally, when you publish an interface you want to publish only the public data. In C++, to publish the interface for a class, you include the class definition in a header file. There is a problem, however; the class definition includes both public and private data.

This can cause problems.

Let's go back to our machine tool example. We had a hardware support module; it's a published interface (header file). For testing, we replaced it with a hardware simulation module that used the same published interface (the exact same header file).

When the interface is a class, you can't do that. That's because your hardware support class will probably have different private members than the simulation support class. Thus you need to keep two header files around, one for the hardware module and one for the simulation.

The public information in these header files must be duplicated, and thus you have all the problems associated with trying to keep two different files up to date and synchronized.

Unfortunately, C++ is not perfect, and the fact that private information must be published is one of its big problems.



In any design, the architect must make a number of trade-offs and compromises. In the case of C++, it was designed to be mostly compatible with the older C language. The designer, Bjarne Stroustrup, also wanted something that could be compiled using the technology of the time (early 1980s).

It was these factors that led him to design classes the way they are. Unfortunately, as a side effect of this design, interface and implementation information were both forced into the class definition.

But given the circumstances under which he worked, Mr. Stroustrup did a brilliant job of creating a new language, in spite of any rough spots which may appear.

Real-World Design Techniques

Over the years, people have developed a number of clever design techniques to help organize their programs. This section will discuss some of the more useful ones.

The Linked List Problem

The linked list problem is actually a C problem, but the various solutions and its ultimate solution in C++ provide a good understanding of the various techniques that can be used to solve a problem.

The code I'm working on now has several linked lists:

- Pending message list
- Running process list
- Keyboard event list
- Idle process list
- Registered connection list
- ... and so on.

There is an insert and delete function for each type of list:

```
insert_msg / remove_msg
insert_run / remove_run
insert_kbd / remove_kbd
insert_idle / remove_idle
insert_connect / remove_connect
... and so on
```

This is a needless duplication of code. There has to be a better way. In C, one solution is to play games with the data. The trick is to define a common linked-list structure:

```
/* C Code */
struct list_head {
    struct list_head *next, *prev;
};
```

This structure is put at the start of each data structure that may be used in a linked list:

```
/* C Code */
struct pending_message_node {
    struct list_head list; /* List info. Must be first */
    struct message the_message; /* The stored message */
};
```

Now we can take advantage of the fact that each node in our pending message list begins with a `list_head`, use casting to turn our `pending_message_node` into a generic node, and use the generic linked list procedures on it:

```
/* C "solution" to a difficult code resue problem */
struct pending_message_node *pending_messages = NULL;
struct pending_message_node *a_new_message;

/* Fill in node */
add_node_to_list(
    (struct list_head *)pending_messages,
    (struct list_head *)a_new_message);
/* Note the C style casts. This is C code */
```

This technique depends on the fact that the compiler lays out the memory for a structure with the first field (in this case `list`) first. This sort of layout is not required by the standard, but almost all compilers do it. (And those that don't cause people who depend on this feature a lot of headaches. They'll be forced to rewrite their code so it doesn't depend on compiler-dependent features.)

The C solution to this problem is pretty good given the limitation of the C language. But the C++ language gives us many more techniques for solving this problem.

The use of C casts is just the poor man's way of doing base and derived classes. The C++ equivalent is:

```
/* Not a good solution. See below */
class list {
private:
    list *next;
    list *prev;
public:
    // Rest of the stuff
};

class pending_message_node: public list {
```

```

    // .... message data
};

```

But there is a problem with this organization. The list and the message have nothing in common. A list is not a refinement of a message, and a message is not a refinement of a list.

Code that wants to process a message and doesn't want to deal with a list item can't deal with a `pending_message_node`. We could write it as:

```

/* Not a good solution still */
class list {
    // ...
};
class pending_message {
    // ...
}
class pending_message_node: public list, public pending_message {
    // Nothing needed here
};

```

This “structure” merely rearranges a badly designed data structure into one that's even more silly.

Ideally we want to organize our information like this:

```

class pending_message {
    // ...
};

class pending_message_list {
    // List stuff
public:
    class pending_message message;
};

```

But now we're back to the problem that started this discussion. We're going to have to create a list for every type of object:

```

class msg_list ....
class keypress_list ...
class event_list ...
class free_block_list ...

```

This means that to properly design our data, we're going to have a lot of almost identical classes. This would be a lot of duplication of effort, except for one thing: templates. The result is that we can write all these classes using one template:

```

template class list<typename data> {
    // List stuff
public:
    data node;
};

```

One last note: it's a little easier that this. We don't have to write the list class ourselves. It's already part of the Standard Template Library (STL):

```
#include <list>

class msg { /* .... */ };
std::list<msg> message_list;

class keypress { /* ..... */};
std::list<keypress> keypress_list;
```

Thus we've taken the long way round to discover that all you need to solve the "list problem" is to use the STL. But it's interesting how the problem can be solved using a language with limited features (C), as well as seeing a number of designs to avoid.

Callbacks

Let's suppose you are writing a text editor. There are three main modules to this program:

- A keyboard module that reads and decodes input
- A buffer/file module that keeps track of the text being edited
- A set of command modules that provide commands that change the text

One of the keyboard module's jobs is to read the keyboard input and call the appropriate command function. The mapping of keys to commands is accomplished through the use of a configuration file (also under control of the keyboard module).

So to do its job, the keyboard module needs to know the names of all the commands and what function to call to execute them.

One way to organize this information is to create a table containing this information and give it to the keyboard processor:

```
struct cmd_info {
    const char *command;
    void (*function)();
}[] cmd_table = {
    {"delete", do_delete},
    {"search", do_search},
    {"exit", do_exit},
    ....
}
```

But this means that the module containing the table must know every command in the system. This way of doing things causes two major problems. First, we have discarded our module hierarchy and invalidated some of the information hiding we have so carefully built up. Now, one module, the command table module, needs to know everything. Second, we have to maintain the thing. Any time any one of the command modules changes, the file containing the command table must change as well.

Figure 26-5 shows our module layout.

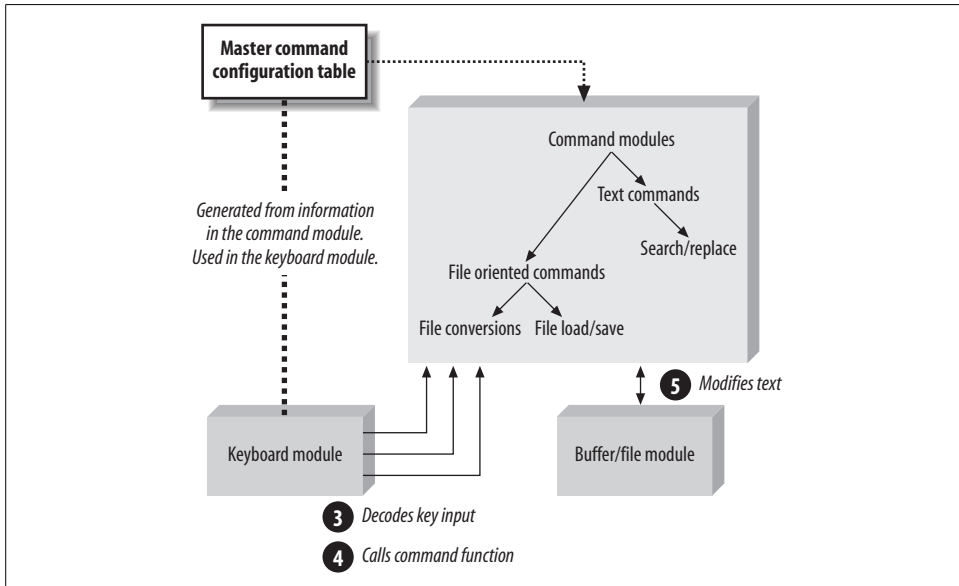


Figure 26-5. Editor module design

A better solution is to build this table at run time. During initialization time, the top command module is told to register all the user-level commands. For example, it may register the “exit” command:

```
keyboard_module::register_command("exit", &do_exit);
```

The top-level command module knows about its submodules and tells them to register their commands. They in turn call the subsubmodules, and so on.

The result is that the command table is built up at run time. Thus, we let the computer keep track of all the commands instead of doing it manually. This is more reliable and easier to do.

Figure 26-6 shows these module interactions.

Thus callbacks let us organize things so that the program configures itself automatically.

Decoupling the Interface and Implementation

Let’s suppose we wish to create a simple database. In it we will store a person’s name and phone number. A class to handle this might look like the following:

```
class phone_book {  
public:  
    void store(const std::string& name, const std::string& number);
```

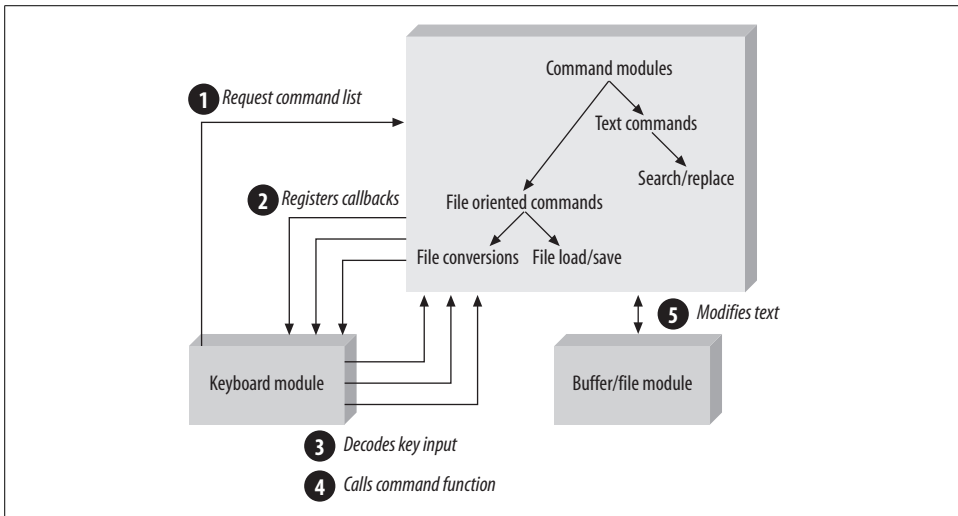


Figure 26-6. New Editor module design

```

    std::string& lookup(const std::string& name);
    void remove(const std::string& name);
    // .. additional member functions
private:
    // Implementation dependent stuff
};

```

This class definition is then stuck in a header file for all to use.

The problem is that we are forced by C++ to put implementation-dependent stuff in the header file. This means that the user knows implementation details that are none of her business. It also means that we cannot provide more than one implementation of our phone book class. (We could play some games with derived classes, but that just moves the problem down a level to the derived class.)

Let's change our definition slightly and write our class as follows:

```

class phone_book_implementation;

class phone_book {
public:
    void store(const std::string& name, const std::string& number);
    std::string& lookup(const std::string& name);
    void remove(const std::string& name);
    // .. additional member functions
private:
    phone_book_implementation *the_implementation;
};

```

Note that we did not define what the `phone_book_implementation` actually is. C++ is perfectly happy to create a pointer to it without knowing anything about it (other than that it is a class of some sort).

It is the job of the `phone_book` constructor to connect the implementation with the interface. In this example, we use a array-based phone book implementation:

```
phone_book::phone_book() {
    the_implementation = new phone_book_array_implementation;
}
```

But there is nothing to prevent us from using a hash-based implementation:

```
phone_book::phone_book() {
    the_implementation = new phone_book_hash_implementation;
}
```

Using this system, we've decoupled the implementation from the interface. This is a good thing (mostly). In this case the caller does not know which implementation was selected.

For example, we could use array-based implementations if we have, say, 1–100 names, a hash for 100–10,000, a small database such as MySQL for 10,000 to 10,000,000, and a commercial database for more than 10,000,000.

What's more, it is possible to switch implementations at run time. For example:

```
phone_book::store(const std::string& name, const std::string& number) {
    number_of_entries++;
    if (need_to_switch_from_hash_to_mysql_database()) {
        phone_book_implementation *new_implementation =
            new phone_book_mysql_implementation;
        copy_database(new_implementation, the_implementation);
        delete(the_implementation);
        the_implementation = new_implementation;
    }
    // .. rest of the function
};
```

As you can see, this design has certain advantages. By decoupling the implementation from the interface, we've gained tremendous flexibility in choosing an implementation.

The design also has some drawbacks, the biggest of which is that it adds an extra layer between the `phone_book` user and the implementation. In most simple programs, this layer is not needed. Also, the use of derived classes in such a situation is no longer a simple matter of C++; instead, it requires some tricky coding. But this does serve to show what you can do with C++ to solve problems creatively.

Conclusion

The best single piece of advice I can give you concerning a program design is:

Do One!

Amazingly, there are a lot of people out there who start coding without thinking about what they are doing beforehand. If you think, then code, the result is much better code. A tenfold improvement can easily be achieved.

Second, show your design to your peers and get them to review it. Experience is one of the best design tools; you have the experience of others.

(The design of this chapter was reviewed by an editor. Its implementation was reviewed by the editor and a set of technical reviewers, one of whom thinks nothing of issuing forth with loud, harsh criticism whenever I do something stupid.)

Finally, C++ gives you a number of tools and techniques for designing your programs. These give you the ability to create a design that is clear, simple, and does the job. A well-designed program is a thing of beauty, so go out there and make the world a more beautiful place.