C++ Hacker's Guide

by Steve Oualline

Table of Contents

Preface

Originally term *hacker* meant someone who did the impossible with very little resources and much skill. The basic definition is “someone who makes fine furniture with an axe”. Hackers were the people who knew the computer inside and out and who could perform cool, clever, and impossible feats with their computers. Now days the term has been corrupted to mean someone who breaks into computers, but in this book we use hacker in its original honorable form.

My first introduction to true hackers was when I joined the Midnight Computer Club when I went to college. This wasn't an official club, just a group of people who hung out in the PDP-8 lab after midnight to program and discuss computers.

I remember one fellow who had taken $10 of parts from Radio Shack and created a little black box which he could use with an oscilloscope to align DECTape drives. DEC at the time needed a $35,000 custom built machine to do the same thing.

There were also some people there who enjoyed programming the PDP-8 to play music. This was kind of hard to do since the machine didn't have a sound card. But someone discovered that if you put a radio near the machine the interference could be heard on the speaker. After playing around with the system for a while people discovered how to generate tones using the interference and thus MUSIC-8 programming system was born. So that the system didn't have a sound card didn't stop hackers from getting sound out of it. This illustrates one of the attributes of great hacks, doing the “impossible” with totally inadequate resources.

My first real hack occurred when some friends of mine were taking assembly language. Their job was to write a function to do a matrix multiply. I showed them how to use the PDP-10's ability to do double indirect indexed addressing[1] which cut down the amount of work needed to access an element of the matrix from one multiply per element to one multiply per matrix.

The professor who taught the assembly class felt that the only reason you'd ever want to program in assembly is for speed, so he timed the homework and compared the results against his “optimal” solution. Every once in a while he'd find a program that was slightly faster, but he was a good programmer so people rarely beat him.

1 The only machines I know of with this strange addressing mode were the PDP-10 and PDP-20. The closest you can come this hack on today's machines involves vectorizing the matrix.

Except when it came to my friends' matrix multiply assignment. The slowest came in at ten times faster than his “optimal” solution. The fastest was so fast that it broke the timing tools he was using. He had to admit it was a neat hack. (After seeing this very strange code, he did something very unusual for a professor: he called my friends to the front of the class, gave them the chalk and had them teach him.)

What makes a good hack? It involves go over, around, or through the limitations imposed by the machine, the compiler, management, security[2] or any thing else.

True hackers develop tricks and techniques designed to overcome the obstacles in front of them and to improve the quality of the systems they work with. These are the true hacks.

This book contains a collection of hacks born out of over forty years of programming experience. Here you'll find all sorts of hacks to make your programs more reliable, more readable, and easier to debug. In the true hacker tradition, this is the result of observing what works and how it works, improving the system, and then passing the information on.

Real World Hacks

I am a real world programmer so this book deals with real world programs. For example, there is a bit of discussion on the care and feeding of C style strings (`char*`). This has angered some of the C++ purist who believe that you should only use only C++ strings (`std::string`) in your programs. That may be true, but in the real world there are lots of C++ programs which use C style strings. Any working hacker has to deal with them.

Idealism is nice, but I work for a living and this book is based on real world, working programs, not the ones you find in the ideal world. So to all the real world hackers out there, I dedicate this book.

2 True hackers only break security to discover weaknesses in the system or to make improvements that the current security policy doesn't allow them to do. They don't break into so they can steal, copy protected information, or spy on other people.

Chapter 1: General Programming Hacks

C++ is not a perfect language. As such sometimes you must program around the limits imposed on you by the language. In this chapter we present some of the simple, common hacks which you can use to make your programs simpler and more readable.

Hack 1: Make Code Disappear

The Problem: Writing code takes time and introduces risk.

The Hack: Don't write code. After all the code that you *don't* write is the easiest to produce, debug, and maintain. A zero line program is the only one you can be sure has no bugs.

A good hacker knows how to write good code. An excellent hacker figures out how to not write code at all.

When you are faced with a problem, sit down and think about it. Some large and complex problems, are really just small, simple problems hidden by confused users and ambitious requirements. Your job is to find the small simple solution and to *not* write code to handle the large confusing one.

Let me give you an example: I was asked to write a license manager which allowed users who had a license key to run the program. There were two types of licenses, those that expired on a certain date and those that never expired.

Normally someone would design the code with some extra logic to handle the two types of licenses. I rewrote the requirements and dropped the requirement for licenses that never expired. Instead we would give our evaluation customers a license that expired in 60-90 days and give customers who purchased the program a license that expired in 2038[3].

Thus our two types of licenses became one. All the code for permanent license disappeared and was never written.

In another case I had to write a new reporting system for a company. Their existing system was written in a scripting language that was just to slow and limited. At the time they had 37 types of reports. With 37 pieces of code to generate these 37 reports.

3 The UNIX **time_t** type runs out of bits in this year.

My job was to translate these 37 pieces of code from one language to another. Instead of just doing what I was told, I sat down and studied what was being done. When my boss asked why I wasn't coding, I told him that I was thinking, a step I did not consider optional.

It turns out that I was able to distill the 37 different reports into just 3 report types. All 37 reports could be generated using these three types and some parameters. As a result the amount of code needed to do the work was cut down by at least a factor of 10.

Remember the code that you never write is the quickest to produces and the most bug free code you'll ever make. Hacking something out of existence is one of the highest forms of hacking.

Producing Lines of Code

I was once tasked with updating a large web based reporting system written in Perl. At this time management decided to measure lines of code written to see how productive its programmers were.

Because of the “design” of the Perl syntax, the difference between bad programmers and good one is amplified. The first week, I cleaned up the obvious inefficiencies and removed a lot of redundant and useless code. My score for that week was about —1,700 lines produced. So the program got smaller even though I added lots of comments and a couple of new features.

For next few weeks I continued to reduce the size of the program. The big change came when I took out the old style, “call function, check for error, pass error up the call change” logic and replaced it with exception based error handling[4]. That change lost us 5,000 lines.

My manager asked me why they should be paying me the big bucks since my *#lines produced / week* was negative. I told them that that was precisely why they were paying me the big bucks. Because it takes a really excellent programmer to produce new features in negative lines of code.

4 The perl module Error implements exceptions.

Hack 2: Let Someone Else Write It

The Problem: Writing code is slow. Write good code is slower, and even if you write good code, you need to spend time debugging it.

The Hack: Build on the work that has proceeded you.

Next to don't do it at all, let someone else do it is the easiest way of programming. There are thousands of tools, programs, and other software out there. One of these might do your job for you. If not, it might almost do the job so all you have to do is download fix it up a little and use it.

One good source for Open Source software is http://www.freshmeat.net. It is a web based database containing a entry for most software project.

Now if you do use Open Source software as a base for your work, you have an obligation to contribute back to the community any enhancements you've created. This lets other people use your work as a base for their programs.

It should be pointed out that some people who are not familiar with how Open Source works are a little frightened by it. They tends to be business men who can't understand how someone would make money writing open source code. The secret is that Open Source is not written by people who want money but by people who want software that works.

And if you want a program that just works, one of the easiest ways of "creating" it is to use other peoples' work as a starting point. That's the hacker spirit: You don't just copy what someone else has done, you push the state of the art forward.

Hack 3: Use the const Keyword Frequently For Maximum Protection

The Problem: You pass a string (`char*`) into a function and the code gets confused because someone accidentally modified the pointer.

The Hack: Tell the compiler that the pointer is not to be changed.

This hack makes use of one of the more difficult to understand concepts of the C++ language, that of **const** and pointers.

We'll start with the declaration:

```
const char* ptr_a;
```


The question is "What does the **const** modify?" Does it affect the pointer or does it affect the data pointed to by the pointer?

In this case the **const** tells the compiler that the character data is constant. The pointer itself can be reassigned.

```
const char* ptr_a;
```

That means that we can reassign the pointer:

```
ptr_a = "A New Value";
```

But you can't modify the data pointed to by the pointer:

```
*ptr_a = 'x'; // ILLEGAL
```

Now let's consider another declaration:

```
char* const ptr_b;
```

In this case the *pointer* is affected by the **const**. The data pointed to is not.

So we can change the data being pointed to:

```
*ptr_b = 'x';           // Legal
```

But we can not change the pointer:

```
ptr_b = "A new string"; // ILLEGAL
```

And of course there's the obvious declaration in which both the pointer and the data are constant:

```
const char* const ptr_c;
```

Now let's go back to our function call. If we are expecting constant data, then let's specify it in the function parameters:

```
void display_string(const char* const the_string);
```

Now any attempt to modify the string will result in a compile time error. And compile time errors are much easier to locate and fix than run time errors.

The const Memory Hack

It's not obvious from the syntax where a **const** keyword affects the pointer or the character. But there is a simple mnemonic trick that may help you remember which is which.

The **const** modifies the element it's nearest. For example:

```
const char* ptr_s;
```

In this case **const** is nearer to `char` than it is to `*`, so the data (`char`) is constant.

In the other case:

```
char* const ptr_t;
```

the **const** is nearer to the `*` than it is to the `char`, so the pointer is constant, not the data (`char`).

Hack 4: Turn large parameter lists into structures

The Problem: Functions that take a large number of parameters are difficult to deal with. Parameters can easily get mixed up.

Although there's no limit on the number of parameter you can pass to a function, in practice more than about six tend to make function calls difficult to use. Consider the following function call to draw a rectangle:

```
draw_rectangle(
    x1, y1, x2, y2, // The corners of the rectangle
    width,          // Width of the line for the rectangle
    COLOR_BLUE,     // Line color
    COLOR_PINK,     // Fill color
    SOLID_FILL,     // Fill type
    ABOVE_ALL       // Stacking order
    "Times",        // Font for label
    10,             // Point size for label
    "Start"          // Label
);
```

This code is an accident waiting to happen. Forget a parameter and the code won't compile. Worse, reverse two parameters and your code may compile but draw the wrong thing.

The Hack: Use a structure to pass a bunch of parameters.

Let's see how that would work for our rectangle function.


```
// Define how to draw the rectangle
struct draw_params my_draw_style;

my_rect.width = width;
my_rect.line_color = COLOR_BLUE;
my_rect.fill_color = COLOR_PINK;
my_rect.fill = SOLID_FILL;
my_rect.stack = ABOVE_ALL;
my_rect.label_font = "Times";
my_rect.label_size = 10;
my_rect.label = "Start";

draw_rectangle(x1, y1, x2, y2, &my_draw_style);
```

Now instead of passing parameters by position they are passed by name. This makes the code more reliable. For example, you no longer have to remember if the line color comes first or the fill color comes first. When you write it as:

```
my_rect.line_color = COLOR_BLUE;
my_rect.fill_color = COLOR_PINK;
```

it's clear which is the line color and which is the fill.

Hacking the Hack: The `draw_params` structure can be used not only for drawing a rectangle but for drawing other shapes as well. For example:

```
draw_rectangle(x1, y1, x2, y2, &my_rect);
draw_circle(x3, y3, radius, &my_rect);
```

It is a good idea to make the default value for any parameter zero. That way, you can set everything to the default using the statement:

```
memset(&my_rect, '\0', sizeof(my_rect));
```

The frees you from having to set values for every item in the structure. For example, to draw a red rectangle using the default width, fill, and label parameters, use the following code:

```
memset(&my_rect, '\0', sizeof(my_rect));
my_rect.line_color = COLOR_RED;
draw_rectangle(x1, y1, x2, y2, &my_rect);
```

If you are using C++, the `draw_params` structure can be made a class. The class can provide internal consistency checking to the user. ("Setting the label to 'foo' with a point size of 0 makes no sense to me.")

Hack 5: Defining Bits

The Problem: You need to define a constant for "bit 5"

Frequently programmers are required to access various bits from a byte. Here's a diagram from a hardware manual for a DLT tape drive:

Bit Byte	7	6	5	4	3	2	1	0
0 -1	(MSB) Parameter Code (LSB)							
2	DU	DS	TSD	ETC	TMC		Rsvd	LP
3	Parameter Length							
4 - 7	(MSB) Parameter Value (LSB)							

You need to define constants to access the various bits in the option byte (Byte 2). One way of doing this to define a hexadecimal constant for each component.

```
// Bad Code
const int LOG_FLAG_DU = 0x80;     // Disable update
const int LOG_FLAG_DS = 0x40;     // Disable save
const int LOG_FLAG_TSD = 0x20;    // Target save disabled
const int LOG_FLAG_ETC = 0x10;    // Enable thres. comp.
```

The problem is that the relationship between 0x40 and bit 6 is not obvious. It's easy to get the bits confused.

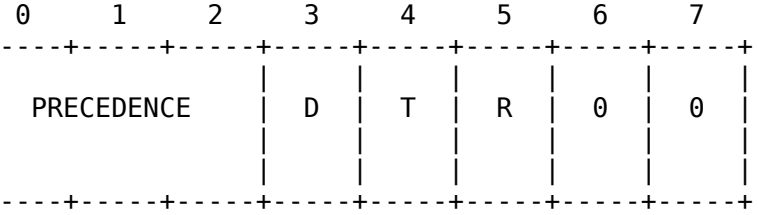
```
// Good code
const int LOG_FLAG_DU = 1 << 7;   // Disable update
const int LOG_FLAG_DS = 1 << 6;   // Disable save
const int LOG_FLAG_TSD = 1 << 5;  // Target save disabled
const int LOG_FLAG_ETC = 1 << 4;  // Enable thres. comp.
```

Now it's easy to see that (1<<4) is bit 4.

Warning: Make sure you know which end is which. In the previous example bit zero is the least significant bit (rightmost bit).

But some people label bit 0 as the most significant bit (leftmost bit). For example the Internet Specification RFC 791 defines the "Type of Service" field as (spelling errors in the original):

```
Bits 0-2:  Precedence.
Bit    3:  0 = Normal Delay,      1 = Low Delay.
Bits   4:  0 = Normal Throughput, 1 = High Throughput.
Bits   5:  0 = Normal Relibility, 1 = High Relibility.
Bit  6-7:  Reserved for Future Use.
```



[Spelling errors in the original.]

We can still use our shift hack to define bits in this way. Only we start with the constant 0x80 and shift to the right.

For example:

```
// Good code
// Delay flag
const unsigned int IPTOS_LOWDELAY = 0x80 >> 3;

// Throughput flag
const unsigned int IPTOS_THROUGHPUT = 0x80 >> 4;

// Reliability flag
const unsigned int IPTOS_RELIABILITY = 0x80 >> 5;
```

Warning: Make sure that you use **`unsigned int`** instead of **`[signed] int`** when defining the constants. Signed integers will cause the sign bit to be replicated in the data yielding unexpected results.

Trivia: The following is the definitions as defined in the Linux standard header file */usr/include/netinet/ip.h:*

```
#define IPTOS_LOWDELAY          0x10
#define IPTOS_THROUGHPUT        0x08
#define IPTOS_RELIABILITY       0x04
```

You may have noticed that they don't use this hack. Without looking at the previous page can you tell which bit is represented by `IPTOS_LOWDELAY`?

Hack 6: Use Bit fields Carefully

The Problem: You want to use bit fields so that you don't have to test, set, and clear bits the hard way. For example:

```
struct timestamp {
    unsigned int flags:4;
    unsigned int overflow:4;
};
```

The Hack: A good hacker treats bit fields with care. There are a number of problems with their use. These include:

1. Order is not guaranteed.

2. Packing is not guaranteed.

3. You really know what can and can't be put in a field. This is especially true when dealing with a bitfield one bit wide as we shall see below.

The C++ standard makes no guarantee where the bits of bit field will end up. In the previous example, the compiler may:

1. Assign the field `flags` to the high bits and `overflow` to the low bits.

2. Assign the field overflow to the high bits and `flags` to the low bits.

3. Ignore the bitfield specification and assign `overflow` and `flags` to different bytes.

The Linux operating systems assumes that you are using the GCC compiler which does pack multiple fields into a single byte. But the ordering of these fields depends on the endianness of the machine.

This results in some strangeness in the header files:

```
struct timestamp {
#if __BYTE_ORDER == __LITTLE_ENDIAN
    unsigned int flags:4;
    unsigned int overflow:4;
#elif __BYTE_ORDER == __BIG_ENDIAN
    unsigned int overflow:4;
    unsigned int flags:4;
#else
# error   "Please fix <bits/endian.h>"
#endif
};
```

There's one other gotcha you have to be concerned about with bit fields. Take a look at the following code:

```
struct flag_set {
   int big:1;
   int bigger:1;
   int biggest:1;
};
// ...
flag_set the_set;
the_set.big = 1;

std::cout << “big flag is “ << the_set.big << std::endl;
```

The output of this program is *not* “big flag is 1”. What is going on?

The problem is that we have a one bit *signed* integer. In a signed integer the first bit is the sign bit. If the first bit the number is negative.

So a single bit signed number can only take two values, 0 and -1.

So the statement:

```
the_set.big = 1;
```

sets the sign bit to 1 making the number negative, setting the field to -1.

Hack 7: Documenting bitmapped variables

The Problem: Bitmapped data is quite common but complex and difficult to use. Such data declarations should really be commented, but unfortunately there's no way for a programmer to draw figures or tables inside a program. It is possible to document things using an external document, but the problem with external documents is that they get out of date easily or get lost.

Ideally the documentation should be embedded in the program as comments. After all, it's difficult to loose ½ a program file.

However, all the nice word processor drawing functions you're used to having when you write a document are missing when you are writing a program. Instead you have to get creative with the mono spaced single font used for writing programs.

Here's one example which uses ASCII art to draw lines from the bits to their description:

```
/*
 * LOG_SELECT parameters byte
 *
 *   +---------- DU (Disable Update)
 *   |+--------- DS (Disable Save)
 *   ||+-------- TSD (Target Save Disable)
 *   |||+------- ETC (Enable Threshold Compression)
 *   ||||++----- TMC (Threshold Met Criteria)
 *   ||||||+---- Rsvd (Reserved)
 *   |||||||+--- LP (List Parameter)
 *   76543210
 */
```

The other method is to use the that they use in the RFC documents. Here's a comment made from an excerpt from RFC 791:

```
/*
 * Bits 0-2:  Precedence.
 * Bit    3:  0 = Normal Delay,      1 = Low Delay.
 * Bits   4:  0 = Normal Throughput, 1 = High Throughput.
 * Bits   5:  0 = Normal Relibility, 1 = High Relibility.
 * Bit  6-7:  Reserved for Future Use.
 *
 *    0     1     2     3     4     5     6     7
 * +-----+-----+-----+-----+-----+-----+-----+-----+
 * |                 |     |     |     |     |     |
 * |   PRECEDENCE    |  D  |  T  |  R  |  0  |  0  |
 * |                 |     |     |     |     |     |
 * +-----+-----+-----+-----+-----+-----+-----+-----+
 */
```

[Spelling errors in the original.]

Copying from a standard like this has the added advantage of faithfully reproducing the information in the standard, thus producing good documentation. And since all you did was copy and past the amount of work required was minor.

The only drawback to copy and paste is that any flaws in the original, such as the spelling error above are also reproduced.

Hack 8: Creating a class which can not be copied

The Problem: You've created a complex class but don't want to create a copy constructor or assignment operator for the class. Besides nobody should be copying any instances of this class anyway.

One solution is to create a copy constructor which aborts the program if it's called:

```
// Works, but not optimal
class no_copy {
    // ...
    public:
        no_copy(const no_copy&) {
            std::cerr <<
               "ERROR: no_copy(no_copy) called" <<
                std::endl;
            abort();
        }
};
```

This works but you'd rather detect the problem at compile time than run time.

The Hack: Declare the copy constructor and assignment operator private.

```
class no_copy {
    // ...
    private:
        no_copy(const no_copy&);
        no_copy& operator = (no_copy&);
};
```

Now let's see what happens when you try and use the copy constructor:

```
no_copy a_var;

// This will result in a compile error
no_copy b_var(a_var);
```


The result is the error message:

```
no_copy.cc:5: error: `no_copy::no_copy(const no_copy&)' is private
no_copy.cc:10: error: within this context
```

Now in actual practice your call to the copy constructor may not be so obvious. What will probably happen is that you'll accidentally call the copy constructor through parameter passing or some other hidden code. But the nice thing is that now when you do call it, you'll discover the problem at compile and not run time.

Hack 9: Creating Self-registering Classes

The Problem: You are writing a program like a image editor that has hundreds of commands. How do you build a master command list.

The Hack: Self registering classes.

The solution is to make each instance of the class register itself. For this example we will define a `cmd` class which defines a named command for our editor. A derived class will be created using this as a base for each command. The derived class is responsible for defining a `do_it` function which performs the actual work.

When a command is created the `cmd` class will register it.

```
        cmd(const char* const i_name):name(i_name) {
            do_register();
        }
```

Naturally we have to unregister it when it's destroyed:

```
        virtual ~cmd() {
            unregister();
        }
```

Remember this is a base class, so we must make the destructor **virtual**.

As hackers we wish to hide as much information as possible from the users. In this case we're going to keep the list of commands and the `do_register` and `unregister` functions entirely within the class.

First the list of commands is declared as a static member variable:

```
    private:
        static std::set<class cmd*> cmd_set;
```


A lot of people don't understand what it means to declare a member variable **static**. A instance of a normal member variable is created when a new instance of a class is created. In other words `a_var.member` is a distinct and different variable than `b_var.member`.

But **static** members are different. For **static** member variable only one instance of the variable is created period. It is shared among instances of the class. So in other words `x_cmd.cmd_set` is the same as `y_cmd.cmd_set`. You can also refer to the variable without an instance of the class at all: `cmd::cmd_set`. (Well you could if we didn't declare it **private**.)

Now let's take a look at our `do_register` function:

```
        void do_register() {
            cmd_set.insert(this);
        }
```

This simply insert a pointer to the current class into the list.

The `unregister` function is just as simple:

```
        void unregister() {
            cmd_set.erase(this);
        }
```

Now comes the fun one, the function we call to execute a command. It is declared as a **static** member function so that we may call it without having a `cmd` variable around.

```
        static void do_cmd(const char* const cmd_name) {
```

Because it is **static** we can call it with a statement like:

```
   cmd::do_cmd("copy");
```

Note: static member functions can only access **static** member variables and global variables.

The body of the function is pretty straight forward. Just loop through the set of commands until you find one that matches, then call the `do_it` member function.

```
        static void do_cmd(const char* const cmd_name) {
            std::set<class cmd*>::iterator cur_cmd;

            for (cur_cmd = cmd_set.begin();
                    cur_cmd != cmd_set.end();
                    ++cur_cmd) {
```



```
                if (strcmp((*cur_cmd)->name,
                                cmd_name) == 0) {
                    (*cur_cmd)->do_it();
                    return;
                }
            }
            throw(unknown_cmd(cmd_name));
        }
```

One of the more interesting things about this system is what happens when you use `class cmd` to declare a global variable. In that case the command is registered *before main in called*.

In other words C++ goes through the program looking for global variables and calling their constructors before starting each program. You have to be careful when doing this though. There is no guarantee concerning the order in which the classes are initialized.

Basically you don't want to create any global variables who's constructor depends on a command being registered.

The complete class is listed below:

```
#include <set>

class cmd {
    private:
        static std::set<class cmd*> cmd_set;

        const char* const name;

    private:
        void do_register() {
            cmd_set.insert(this);
        }
        void unregister() {
            cmd_set.erase(this);
        }

        virtual void do_it(void) = 0;
    public:
        cmd(const char* const i_name):name(i_name) {
            do_register();
        }
        virtual ~cmd() {
            unregister();
```



```
        }

    public:
        static void do_cmd(const char* const cmd_name) {
            std::set<class cmd*>::iterator cur_cmd;

            for (cur_cmd = cmd_set.begin();
                    cur_cmd != cmd_set.end();
                    ++cur_cmd) {
                if (strcmp((*cur_cmd)->name,
                                cmd_name) == 0) {
                    (*cur_cmd)->do_it();
                    return;
                }
            }
            throw(unknown_cmd(cmd_name));
        }
};
```

A smart hacker might note that we could have used a `std::map` to hold our command list. After all a `std::map` takes a key and value pair and would eliminate our lookup loop in `do_cmd`.

But the extra syntax needed to make a **`std::map`** work would get in the way of the point of this hack. So while the class is not as efficient code wise, it is very efficient book wise.

Hack 10: Decouple the Interface and the Implementation

The Problem: C++ forces you to expose the implementation details when you define a class.

Let's take a look at a typical class definition for a dictionary class. This class lets you define a list of word pairs called the *key* and the *value*. It then lets you use the *key* to lookup the *value.*


```
class dictionary {
    private:
        // Lots of stuff that the user doesn't need to
        // worry about.
        // (And this is the problem)
    public:
       // ... usual constructor / destructor stuff
       void add_pair(const std::string& key,
            const std::string& value);
        const std::string& lookup(const std::string& key);
};
```

The problem is that this gives anyone who uses this class access to some of the private implementation details of the class.

The Hack: Hide the implementation details through the use of an implementation class.

```
class dictionary {
    private:
        dictionary_implementation* implementation;
    public:
```

Since we don't have to define `dictionary_implementation` in the header file this effectively hides the implementation and separates the interface and the implementation.

Hacking the Hack: There are several ways of implementing a dictionary. For example, if we are dealing with about 10-100 words, we could implement the dictionary using an array.

For 100-100,000 entries we could use a dynamic list. For over 100,000 the code can make use of an external database.

But what's nice about defining a dictionary in this way is that the class can change the implementation on the fly as conditions change. For example, the class can start with an array. When the number of entries grows to more than 100 it can switch to a list based implementation:


```
void dictionary:: add_pair(const std::string& key,
            const std::string& value) {
    if ((implementation->type() == ARRAY_IMPLEMENTATION)&&
        (implementation->size() >= MAX_ARRAY)) {
        dictionary_implementation* new_implementation =
            new dictionary_as_list(implementation)
        delete implementation;
        implementation = new_implementation;
    }

    // ... same thing for    list -> database

    implementation->add_pair(key, value);
```

Thus we've not only hidden the dictionary implementation, but we've made it able to dynamically reconfigure itself depending on data load.

Hack 11: Learning From The Linux Kernel List Functions

The Problem: There are lots of ways of creating a linked list. There are only a few good ones.

The Hack: The Linux kernel's linked list implementation. (See the file */usr/include/linux/list.h* in any kernel source tree.)

There a large number of things we can learn from this simple module.

First since linked lists are a simple and common data structure it makes sense to create a linked list module to handle them. After all things can get confused if everyone implements his own linked list. Especially if all the implementations are slightly different.

Lesson 1: Code Reuse.

The way the linked list is implemented is very efficient and flexible. You can actually have items that are put on multiple lists.

Lesson 2: Flexible design.

The list functions are well documented. The header files contains extensive comments using the *Doxygen* documentation convention (See Hack 65).

Lesson 3: Share your work. Document it so others can use it.

There's there's a mechanism in place to help persuade people to use this implementation and to avoid writing their own. If you submit a kernel patch containing a new linked list implementation you will be “politely”[5] told to use the standard implementation. Also your code won't get in the kernel until you do.

Lesson 4: Enforcement of standard policy. Mostly through peer pressure.

So by looking at this implementation of a simple linked list we can learn something. Which leaves us with our final lesson:

Lesson 5: A good hacker learns by reading code written by someone who knows more about this type of programming that you do.

5 The term “polite” has a different meaning when dealing with people who frequent the kernel mailing lists.

Chapter 2: Safety Hacks

Some programmers code first and think about safety later. These are the people who spend the first six months of the career writing code and the next five years creating security patches.

Let's suppose you are standing at the top of a very tall cliff. You need to get to the bottom. Would you

1. Start moving immediately and jump off the cliff because that's the fastest way to get to the bottom. (You'll worry about safety after you start your jump.)

2. Analyze the terrain to discover the best method of getting you to the bottom in one piece.

If you answered #1, then you code like most of the programming drudges out there out there. It's the fastest answer you can give if you have absolutely no regard for safety at all.[6]

A true hacker's answer is “I'd think for moment and figure out the fastest safe route. Then I'd mark it as I went down to help those who follow me.”

C was never designed for safety. C++ builds on this foundation to create a more complex and unsafe language. Yet there are hacks which can use to make your programs safer. In other words your program will crash less, and even if they do crash, the problem will be easier to find.

Hack 12: Eliminate Side Effects

The Problem: C++ let's you use operators like ++ and -- to make your code very compact. It also can be used to create code that generates ambiguous results, as well as being unreadable.

The Hack: Write statements that perform one operation only. Don't use ++ and -- except as standalone statements.

6 If you think that no programmer would really program this way, just look at the design decisions made by a certain major commercial operation system. These people have to issue weekly security patches to keep up with the safety problems that they themselves introduced because they decided to code and think in *that* order.

For example, is the result of the following code:

```
i = 0;
// Bad code
array[i++] = i;
```

There are two sub-expressions involving **i**. These are:

1. **i++**
2. **i**

The compiler is free to execute them in any order it wants to. So what compiler you are using and even what compilation flags are used can affect the result.

There is no reason for trying to keep everything on one line. You computer has lots of storage and a few extra lines won't hurt things. A simple, working program is always better than a short, compact, and broken one.

So avoid side effects and put **++** and **--** on lines by themselves. If we rewrite the previous example as:

```
i++;
array[i] = i;
```

The order of the operations is clear not only to the compiler but to anyone reading the code.

Hack 13: Don't Put Assignment Statements Inside Any Other Statements

The Problem: It's a classic mistake: using = instead of ==.

Code like:

```
    if (i = getch()) {
        // Do something
    }
```

The Hack: Never include an assignment statement inside any other statement. (Actually never include any statement inside any other statement, but this one is so common it deserves its own hack.)

The reason for this is simple: You want to do two simple things right one at a time. Doing two things at once in a complex, and unworkable statement is not a good idea.

This hack flies in the face of some common design patterns. For example:

```
// Don't code like this
while ((ch = getch()) != EOF) {
    putchar(ch);
}
```

Following this safety rule, our code looks like:

```
// Code like this
while (true) {
    ch = getch();
    if (ch == EOF)
        break;
    putchar(ch);
}
```

Now a lot of people will point out that the first version is a lot more compact. So what? Do you want compact code or safe code? Do you want compact code or understandable code? Do you want compact code or working code?

If we take off the requirement that the code works, I can make the code much more compact. Because most people value things like code that is safe and working, it is a good idea to use multiple simple statements instead of a single compact one.

This hack is designed to keep things simple and constant. As a hacker we know you are a clever programmer. But it takes a very clever programmer to know when not to be clever.

Hack 14: Use const Instead of #define When Possible

The Problem: The **#define** directive defines a literal replacement. The pre-processor is its own language and does not follow the normal C++ rules. That can lead to some surprises.

For example:

```
// Code has lots of problems (don't code like this)
#define WIDTH 8 – 1  // Width of page – margin

void print_info() {
    // Width in points
    int points = WIDTH * 72;

    std::cout << “Width in points is “ <<
```



```
            points << std::endl;
```

Question: What's **WIDTH**? If you answered 7 you got it wrong. The pre-processor is a very literal program. The value of **WIDTH** is literally **8 – 1**.

But everyone knows that 8 – 1 is 7 right? Not everyone. C++ does not. Especially when used in a expression. The line:

```
    int points = WIDTH * 72;
```

is translated by the pre-processor into:

```
    int points = 8 - 1 * 72;
```

As a result, the value of **points** is not what the programmer intended.

The Hack: Use **const** instead of **#define** whenever possible.

If we had defined width as:

```
static const int WIDTH = 8 – 1;  // Width of page – margin
```

then our calculations would be correct. That's because we are now defining **WIDTH** using C++ syntax, not pre-processor syntax.

Using **const** has another benefit. If you make a mistake in the **#define** statement, the problem may not show up until you actually use the constant. The C++ compiler performs syntax checking on **const** statements. Any syntax problems with these statements show up immediately and you don't have to guess where the problem occurred.

Hack 15: If You Must Use #define Put Parenthesis Around The Value

The Problem: There are some times that you just can't use **const**. How do you avoid problems like the one shown in the previous hack?

You might ask why can't we just use **const**? The answer is that sometimes you need to create a header file that's shared with a program in another language. For example the *h2ph* program that comes with Perl understands **#define**, but not **const**.

The Hack: Always enclose **#define** values in ().

For example:

```
#define WIDTH (8 – 1)  // Width of page – margin
```

Now when you use this in a statement like:


```
    int points = WIDTH * 72;
```

you get the correct value.

Hack 16: Use inline Functions Instead of Parameterized Macros Whenever Possible

The Problem: Parameterized macros can cause unexpected things to happen.

Consider the following case:

```
#define SQUARE(x) ((x) * (x))

int i = 5;
int j = SQUARE(i++);
```

What's the value of **j**? The answer is that it's compiler dependent. And the value of **i** is definitely not 6. Why? Just look at the code after the macro is expanded:

```
int j = ((i++) * (i++));
```

From this we can see that **i** is increment twice. Also since the order of the operations is not specified by the C++ standard, the actual value of **j** is compiler dependent.

Note: We violated Hack 12 in this example. This is another example of why that hack is important.

The Hack: Use **inline** functions instead of **#define** whenever possible.

Let's see how our code would look with an **inline** function:

```
static inline int square(int x) {
    return (x * x);
}
```

Now when this function is called two things happen: **j** gets the correct value and **i** is incremented once. In other words, the code behaves just like it is written. And having something look and act the same way is a beautiful hack.

Note: Someone I'm sure is going to point out that the macro works for any type and the **inline** function only works for integers. This problem is easily solved by making the **inline** version a function **template**.

Hack 17: If You Must Use Parameterized Macros Put Parenthesis Around The arguments

The Problem: The way the pre-processor handles parameterized macros can sometimes lead to incorrect code. For example:

```
// Don't code like this
#define SQUARE(x) (x * x)
```

So what's the value of **i** in the following statement:

```
int i = square(1 + 2);
```

Should be 9. But the statement expands to:

```
int i = (1 + 2 * 1 + 2);
```

which gives us 5 not 9.

The Hack: Put parenthesis around every place you use a parameter in a parameterized macro. For example:

```
#define SQUARE(x) ((x) * (x))
```

Now our expanded assignment statement looks like:

```
int i = ((1 + 2) * (1 + 2));
```

and we'll get the right answer.

Note: This does not solve the increment problems shown in Hack 16. This hack should be only used if you absolutely must use parameterized macros and can't use **inline** functions. (See also Hack 12 for help avoiding the increment problem.)

Hack 18: Don't Write Ambiguous Code

The Problem: Consider the following code:

```
if (a)
    if (b)
        do_something();
  else  // Indentation off
     do_something_else();
```

Which **if** does the **else** go with?

1. It goes with the first **if**.
2. It goes with the second **if**.

3. If you don't write code like this you don't have to worry about stupid questions.

The Hack: The hacker's answer is obvious the third one. Hackers know how to avoid trouble before it starts. So if we never get near nasty code we don't have to worry about how it works. (Unless we have to deal with legacy code written by non-hackers.)

Always include {} when there's any ambiguity in your code. The previous example should be written as:

```
if (a) {
    if (b) {
        do_something();
    } else {
        do_something_else();
    }
}
```

From this code it's clear which **if** the **else** belongs to.

Being obvious is an important part of coding safely. There's enough confusion and chaos in programming already without having someone add to it by exploiting obscure elements of the C++ syntax.

A good hacker knows how to keep things simple, obvious, and working.

Hack 19: Don't Be Clever With the Precedence Rules

The Problem: What's the value of the following expression?

```
i = 1 | 3 & 5 << 2;
```

Most people would answer “I don't know.” Hackers would answer “I don't know,” then write a short test program to find out the answer. (And I'm not going to spoil your fun by putting the answer in here.)

But the problem is unless you are really into the C++ standard and have memorized the 17 operator precedence rules you can't tell what this code is doing.

The Hack: Limit yourself to two precedence rules:

1. Multiple and Divide come before addition and subtraction.

2. Slap parenthesis around everything else.

Consistency and simplicity are key to safe programming. The less you have to think and make decisions the less you can make the wrong decision. A good hacker will produce a set of rules and procedures so he can do things consistently and right. Another way of saying this is a good hacker does a great deal of thinking about things so he doesn't have to do a great deal of thinking.

The simplified precedence rules are one example of this. Almost no one remembers the official 17, but remembering the simplified two is simple.

Applying our hack it's easy to figure out the result of the following expression:

```
i = (1 | 3) & (5 << 2);
```

(I know it's not the same result, but this is what the programmer intended in the first place.)

Hack 20: Include Your Own Header File

The Problem: It is possible for a function to be defined one way in a header file and the other in the code.

For example:

square.h

```
extern long int square(int value);
```

square.cpp

```
int square(int value) {
    return (value * value);
}
```

Note: C++ is only *partially* type safe. The *parameters* to a function are checked across modules, the *return values* are not.

So what happens when this function is called? The `square` function computes a number and returns the result, a 32 bit integer[7]. The caller knows that the function returns a 64 bit integer.

Since 32 bit return values and 64 bit return values are returned in different registers, the calling program gets garbage. What's worse the poor maintenance programmer is let wondering how a function like `square` which is to simple to fail, is actually failing.

7 We assume we are on a machine where an **int** is 32 bits.

The Hack: Make sure each module includes it's own header file. If the *square.cpp* file began with:

```
#include “square.h”
```

the compiler would notice the problem and prevent you from compiling the code.

And professional programmers know that it's 10,000 times easer to catch an obvious problem at compile time than it is to locate a random value error in a running program.

Hack 21: Synchronize Header and Code File Names

The Problem: The previous hack tells us that each module should include it's own header file. How can we make that as simple as possible.

The Hack: Always use the same name for both the header file and and the code file. So if the header is *square.h* the code file will be *square.cpp*. When you have a rule like this you don't have to think and avoiding extraneous decision making will help make your code more reliable.

But now let's suppose we have three files *square.cpp, round.cpp,* and *triangle.cpp*. These will be compiled and combined to form a library *libshape.a*. Ideally we would like to supply the user with a single header file so he doesn't have to know or understand our module structure. What do we do?

If we supply him with a *shape.h* file we fulfill our simplicity requirements, but we violate our naming rules.

The answer is that we can do both, provide a single interface file to the user and follow our naming rules. We start by creating three header files for our three modules: *square.h, round.h,* and *triangle.h*. Next we create an interface file for the user, *shape.h* which contains:

```
#include <shape-internal/square.h>
#include <shape-internal/round.h>
#include <shape-internal/triangle.h>
```

The nice thing about this system is that our header files mirror our object files. We assemble a bunch of object files into a library and our group header *shape.h* assembles a bunch of header files into a master header for the user.

One of the best forms of hacking is to simplify things. A good hacker does a lot of thinking and design so he doesn't have to do a lot of thinking or design.

Hack 22: Never Trust User Input

The Problem: Users type in bad things.

Most users type in bad things because they don't understand the software or understand it's limitation. They can look at a prompt like:

```
Enter a user name (5 characters only):
Do not type in more than 5 characters.  It won't work.
Five character is the limit no more.
Absolutely no more than 5 characters please.
User name:
```

and see an open invitation to type in fifty-five characters.

Stupid users enter bad data. Smart users who think they know more than the computer enter bad data. Average users mistype things and enter bad data. I think we can set the pattern here.

What's worse is the malicious user who enter bad data in an effort to crash the system or bypass security. Two classic attack vectors are the stack smashing attack and the SQL injection technique.

In stack smashing attack the user attempts to overflow the input buffer. If he inputs enough data he can overwrite the return address in the stack and trick the computer in executing arbitrary code.

To protect against stack smashing attack always check the length of user input to make sure that the limit is obeyed. If you are using C style I/O this means using *fgets* instead of *gets*. (See Hack 23 below.)

SQL Injection attacks involve the user submitting badly formed data in hope that it's executed in an SQL query. For example, the following SQL code updates a user E-Mail address:

```
UPDATE user_info SET email = 'fred@whatever.com'
    WHERE user = 'Fred';
```

Things go just fine if the user tells you his name is "`Fred`". But a malicious user can play games with the user name. For example, let's suppose he gives us a user name of "`F';SELECT user, password FROM user_info;`" Now our SQL command is:

```
UPDATE user_info SET email = 'fred@whatever.com'
    WHERE user = 'F';SELECT user, password FROM user_info;
```

Better written as:


```
UPDATE user_info SET email = 'fred@whatever.com'
    WHERE user = 'F';
SELECT user, password FROM user_info;
```

The **SELECT** statement will return to the user all the user names and passwords in the database.

Good DB Security Practices

The database schema in this example illustrates a poor database design. You never should sort sensitive information (password, social security number, credit card numbers) in any database accessible directly from the Internet.

Such information should be kept in a dedicated secure computer which allows very limited access from your own computers and no access from any else. It should be locked up tight. Also the data itself should be encrypted with a key that must be entered on the console of the machine at boot time. Only under extremely limited circumstances should unencrypted data be transmitted.

The client / server connection should also be locked down as well. The client should never be able to ask the database for the password. The only thing it should be able to do is to ask “Is this password correct?” And that question should be transmitted over an encrypted link for added security.

Looking up this data this way is not foolproof, but it does keep out most of the bad guys.

And by the way storing sensitive data on a laptop or portable drive, especially credit card numbers and social security numbers is really, really stupid. Do not store sensitive information on portable devices and don't leave such devices in places like a hotel room where they are easy to steal.

Also any sensitive data on your laptop should be protected by a good encryption system.

To prevent SQL injection attacks you should validate all the characters supplied by the user. Here's an example of how *not* to do it:


```
// Bad code
bool validate_name(const char* const name) {
    for (int i = 0; name[0] != '\0'; ++i) {
        if (name[i] == '\'')
            return (false);
    }
    return (true);
}
```

Why is this code bad? Because it only checks for a bad character. Actually in SQL there are more bad characters out there. *You shouldn't check to make sure that the input does not contain bad character, you should make sure that everything is good.*

```
// Good code
bool validate_name(const char* const name) {
    for (int i = 0; name[0] != '\0'; ++i) {
        if (! isalnum(name[i]))
            return (false);
    }
    return (true);
}
```

It is much more secure to only good stuff than to exclude bad stuff. That's because if you make a mistake and make your “good stuff” definition too restrictive you don't cause a security hole in the program. If you make a mistake in a “bad stuff” definition, bad things could get through.

Remember, just because you're paranoid, it doesn't mean they aren't out to get you.

Hack 23: Don't use gets

By now almost everyone knows the all the security and reliability problems that can occur with `gets`. But it's included here for historical reasons as well because it's a very good example of bad programming.

Let's look at all the problems with the code:

```
// Really bad code
char line[100];
gets(line);
```

Because gets does not do bounds checking a string longer than 100 characters will overwrite memory. If you're lucky the program will just crash. Or it might exhibit strange behavior.

But this code is also a security problem. A attacker can create a carefully constructed string which overwrites the stack and let's the bad guy execute any code he wants to.

The `gets` function is so bad that the GNU *gcc* linker issues a warning whenever it's used.

```
/tmp/ccI5WJ5m.o(.text+0x24): In function `main':
: warning: the `gets' function is dangerous and should not be used.
```

The Hack: Use `fgets` instead

```
// Good code
char line[100];
fgets(line, sizeof(line), stdin);
```

The `fgets` call will not get more data than the variable can hold. This prevents attackers from executing a stack smashing attack.

Hack 24: Flush Debugging

The Problem: Buffering of output can lead to unexpected results

For example:

```
std::cout << “Doing unrelated stuff” << std::endl;
do_stuff();

std::cout << “Doing divide ... “;
i = 1/0; // Divide by zero
std::cout << “complete\n”;
```

When runs this program prints (on some systems):

```
Doing unrelated stuff
Floating point exception[8]
```

From this output we can see that the problem is “obviously” in the **`do_stuff`** function. What's going on?

The problem is that output is being buffered. So the string **`“Doing divide ...”`** goes into the buffer, then the program crashes with a divide by zero error, and the output is never displayed. This is where the confusion comes from.

8 Why an *integer* divide by zero causes a *floating point exception* is a question that this book does not deal with.

The Hack: Make sure that when debugging that the buffer is flushed after every output.

There are several ways of doing this. The first is to explicitly flush every statement:

```
std::cout << “Doing divide ... “ << std::flush;
```

This works, but you have to remember to do the flush for every statement.

The other way is to set the **`unitbuf`** flag which tells C++ to flush after every output operation. This only has to be done once at the top of your program.

```
std::cout << std::unitbuf;
// From now on everything is automatically flushed
```

In C the same thing can be accomplished by setting the **`_IONBV`** flag using the **`setvbuf`** function:

```
static char buf[512];   // Buffer for standard out
setvbuf(stdout, buf, _IONBV, sizeof(buf));
```

Being aware of what's going inside the program is very useful to a hacker. Sometimes the compiler, library or the machine will do strange things to you. But knowing the internals is only half the battle. Knowing how to get around the internal limitations of the system is the mark of a good hacker.

Hack 25: Protect array accesses with assert

The Problem: C++ does not do bounds checking.

For example the following code with compile and execute just fine. It will also corrupt memory:

```
// Bad code
int data[10];
// ...
int i = 11;
data[i] = 5;  // Memory corruption
```

The Hack: Use `assert` to check all array accesses.

For example:

```
// Better code
#include <cassert>
```



```
int data[10];
// ...
int i = 11;
assert((i >= 0) && (i < 10)); // Good example
                              // Rotten implementation
data[i] = 5;
```

This works but there is a problem with. We've used the constant 10 in two places. (The declaration and the assert.) It would be easy for someone to change one and not the other.

One solution to this is to use named constants.

```
// Better code
#include <cassert>
const int DATA_SIZE = 10;
int data[DATA_SIZE];
// ...
int i = 11;
assert((i >= 0) && (i < DATA_SIZE)); // Better,
                                     // but not best
data[i] = 5;
```

Ideally we don't want to have to use even named constants if we can help it. It is possible to create a bounds checking `assert` using the `data` variable alone:

```
assert((i >= 0) && (i < sizeof(data) / sizeof(data[0])));
```

So what is going on here? The expression `sizeof(data)` returns the number of bytes in the array `data`. But we need the number of elements in the variable, not the number of bytes.

The solution is to divide the number of bytes in the array by the number of bytes in the first element. The result is an expression which gives us the number of elements.

```
sizeof(data) / sizeof(data[0])
```

Now being true hackers, we don't want to have to write the same expression over and over again, so let's create a macro to make our life easier.

```
/*
 * assert_bounds(var, index) - Make sure an index is in
 *       bounds.
 *
 * Warning: This only works is var is a array variable and
 *    not a pointer.
```



```
 */
#define assert_bounds(var, index) \
   assert((index >= 0) &&
     (index < (sizeof(var) / sizeof(var[0]))));
```

Because we're such nice programmers we've documented our macro and even included a warning describing its limitations.

Array overflows are one of the most common programming errors and are extremely frustrating to try and locate. This code doesn't eliminate them, but it does cause buggy code to abort early in a way that makes the problem tremendously easier to find.

Warning: The assert statement is not guaranteed to abort your program. For example, consider the statement:

```
assert(false);
```

This statement “obviously” will cause the program to abort because the assert is always false. If the program is compiled with `NDEBUG` defined, all the asserts are compiled out. In other words, if `NDEBUG` is defined the previous statement does nothing.

A Real System Crash

You should only use `assert` in programs where aborting is an acceptable behavior. In most cases when a program crashes it is an annoyance for the user but not a disaster. That is not always the case.

In 1996 some people started running a program on an upgraded hardware platform. As a result of the hardware upgrade the program ran longer than expected and one of the counters overflowed. Since this was ADA code, it triggered an exception.

The exception was not caught so the system executed the default exception handler and halted the processor. This was not a good thing to do.

The hardware platform that had been upgraded was the Ariane 4 rocket to the Ariane 5 rocket. The computer in question was tasked with keeping the rocket pointed into the air.

Technically the rocket didn't crash. The launch director blew it up when it started to head for the ground.

The cost of that bug was estimated to be about $500,000,000 (US).

Hack 26: Use a Template to Create Safe Arrays

The Problem: You don't want to have to add asserts every time you access an array.

The Hack: Hide the code inside a template. Then the work is done for you automatically:

```
#include <cassert>

template<typename array_type,
         unsigned int size> class array {
    private:
        static const unsigned int ARRAY_SIZE = size;
        array_type data[ARRAY_SIZE];
    public:
        array(void) {};
        array(const array& other_array) {
            memcopy(data, other_array.data, sizeof(data));
        };
        ~array() {};
        array& operator = (const array& other_array) {
            memcopy(data, other_array.data, sizeof(data));
            return (*this);
        };
    public:
        array_type& operator[](int index) {
            assert(index >= 0);
            assert(index < ARRAY_SIZE);
            return (&data[index]);
        }
};
```

This class has several nice features. The first is that the copy constructor and the assignment operator are implement in a way that allows for copying the array.

But the big advantage of the code is the function which handles the access to an element in the array. It includes `assert` statements which prevent you from overflowing the array:


```
        array_type& operator[](int index) {
            assert(index >= 0);
            assert(index < ARRAY_SIZE);
            return (&data[index]);
        }
```

Another feature of this template is that it doesn't provide a way of converting an array into a pointer. It is up to you whether or not you consider this a feature or a bug.

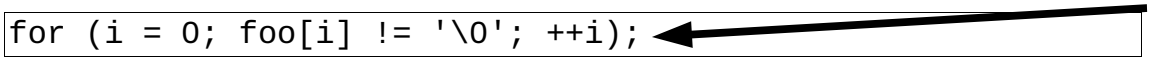
Hack 27: When Doing Nothing, Be Obvious About It

The Problem: The following code is confusing.

```
// Very bad coding style
int i;
for (i = 0; foo[i] != '\0'; ++i);
std::cout << “Result “ << i << std::endl;
```

At first glance it seems that someone failed to indent the program properly. The `std::cout` line should be indented as it is part of the **for** statements.

But on close inspection you can see that the program is indented correctly. There a tiny semicolon at the end of the **for** loop:



This semicolon is almost completely invisible. It would be nice to make it more visible.

The Hack: Do nothing quietly. Always put something in to say that you're doing nothing.

```
// Almost adequate coding style
for (i = 0; foo[i] != '\0'; ++i)
    /* Do nothing */;
```

This is better, but we can improve on it. The **continue** keyword tells C++ to start the loop over again. It can be placed inside our empty loop to provide a more meaty statement for our eyes:

```
// Good coding style
for (i = 0; foo[i] != '\0'; ++i)
    continue;
```


Hack 28: End Every Case with break or /* Fall Through */

The Problem: In the following code did the programmer intend for the STATE_ALPHA case to fall through or did he make a mistake? In other words is that fact that STATE_ALPHA calls `do_alpha` and `do_beta` intentional or an error?

```
// Rotten code
switch (state) {
    case STATE_ALPHA:
        do_alpha();
    case STATE_BETA:
        do_beta();
        break;
// ....
```

From this code it's impossible to tell what the programmer intended.

The Hack: Be obvious about what you do. If you intend for one case to fall through to another indicate it with a comment like `// Fall Through`.

```
// Decent code
switch (state) {
    case STATE_ALPHA:
        do_alpha();
        // Fall through
    case STATE_BETA:
        do_beta();
        break;
// ....
```

After all being a hacker means that you have to be clever, not that you have to be sneaky.

Hack 29: A Simple assert Statements For Impossible Conditions

The Problem: What do you do when you detect an internal error.

You could do an `assert(false);` but that doesn't give you much information when the program dies.

The Hack: Put a message in your `assert` statements. For example:

```
if (this_is_not_possible) {
    assert("INTERNAL ERROR: Impossible condition" == 0);
```



```
    abort();
}
```

The code `"string" == 0` compares the address of the string against 0. They should not be zero and the `assert` fails. Since a failing assert prints the condition that failed, now when your program dies, you will get a nice error message telling you what happens.

But if the program dies when the `assert` statement fails, why put in the abort? because it's possible to use compile time switched (`-DNDEBUG`) to compile *out* the assert statements.

So what this code really says, is die with a nice error message. And if you're still alive, just die.

Hack 30: Always Check for The Impossible Cases In switches

The Problem: A variable can only contain vowels. What do you do if it has the value `'q'`?

Lousy Riddle Time

Q: What time is it when you clock strikes 13?

A: Time to get a new clock.

In our example here, a value of `'q'` indicates an internal error. After all its "impossible" for us to get a value of `'q'`. But that doesn't stop us from getting one.

Someone upstream blew it. There are two major rules of programming safety:

1. Never trust data created by code you didn't write.
2. Never trust data created by code you did write.

A good dose of paranoia is very healthily when it comes to creating safe program. After all just because your paranoid, doesn't mean that the system isn't out to get you.

The Hack: Make the **switch** statement take care of everything - even the stuff that can't exist.

Switches if something can have only have a limited number of values, you should always add a **default** clause to catch things that fall out of range:

```
switch (vowel) {
    case 'e':
        ++e_count;
        break;
    case 'i':
        ++i_count;
        break;
    case 'a':
    case 'o':
    case 'u':
        // Ignore these values
        break;
    default:
        assert("INTERNAL ERROR: Vowel not legal" == 0);
        abort();
}
```

One thing to notice about this code, we clearly indicate what vowels are to be ignored by the statement:

```
        // Ignore these values
```

After all, the idea is to not only be safe but to be clear and obvious as well.

Hack 31: Create Opaque Types (Handles) Which can be Checked at Compile Time

The Problem: You are creating an API which uses a lot of handles. For example, font handles, graphics handles, window handles, color handles, and so forth.

One solution is to create a "different" type for each handle:

```
// Dangerous code
typedef short int font_handle;
typedef short int graphic_handle
typedef short int window_handle;
typedef short int color_handle;
```

The following code shows these types in operation:

```
font_handle the_font = find_font("Times", 10, "Bold");
color_handle the_color = find_color("Light Red");

draw_text(the_font, the_color);
```


But there is a problem with doing things this way. Both `font_handle` and `color_handle` have the same basic type. So the compiler will not complain if you do the following:

```
// Parameters backwards
draw_text(the_color, the_font);
```

Ideally we would like a solution where the compiler will complaint if we mix the handles. The other requirement we have is that the handle take up as little space as possible, preferable two bytes.

The Hack: Use a small structure to hold the handles.

Let's take a look at the following handle declarations:

```
// Safe code
typedef struct {
    short int handle;
} font_handle;

typedef struct {
    short int handle;
} graphic_handle

typedef struct {
    short int handle;
} window_handle;

typedef struct {
    short int handle;
}  color_handle;
```

So we've replaced a two byte integer with a two byte structure. What's the big deal? The big deal is type checking. Now the handles have a different structure and no common base type.

As a result, the compiler will do type checking. In other words the following is legal:

```
draw_text(the_font, the_color);  // Correct and legal
```

and the following is not:

```
draw_text(the_color, the_font); // Incorrect and illegal
```

Note all the advantages of the handle system are still in places. Handles are still small opaque entities, but now they are type checked as well. Thus we we have a hack that gets around C++'s week types and turns them into stronger ones.

Hack 32: Using sizeof When Zeroing Out Arrays

The Problem: Using named constants in `memset` calls will fail if the constants get out of sync.

For example:

```
// Bad code
const int CONNECTION_INFO_SIZE = 1OO;
char connection_info[CONNECTION_INFO_SIZE];

const int SIMPLE_CONNECTION_INFO_SIZE = 3O;
char secure_connection_info[SIMPLE_CONNECTION_INFO_SIZE];

//...
// Zero all connection information
memset(connection_info, '\O',
                                CONNECTION_INFO_SIZE);

// Mistake
memset(secure_connection_info, '\O',
                                CONNECTION_INFO_SIZE);
```

The last statement uses the wrong constant and zeros out the the entire array and then some corrupting memory. Memory corruption is one of the most difficult problems to debug because it can cause strange failures in parts of the code far from the initial problem.

The Hack: Always use `sizeof()` to determine the size of a structure

So the proper way of handling doing a `memset` calls is:

```
memset(array, '\O', sizeof(array));
```

No matter how you change the size and base type of the array the `sizeof()` operator will return the correct size and the `memset` call will do the right thing.

Hack 33: Use sizeof(var) Instead of sizeof(type) in memset Calls

The Problem: Using `sizeof(type)` in a `memset` call is unsafe.

Some programmers consider the following good programming practice:

```
//Not a good idea
struct data_struct {
   int i1, i2;
};

data_struct* data_ptr;
data_ptr = new data_struct;
//....
//Bad code
memset(data_ptr, '\0', sizeof(data_struct));
```

A problem can occur if you modify the code to and modify what `data_ptr` is pointing to.

```
// This code contains a mistake
struct data_struct {
   int i1, i2;
};
struct data_struct_improved {
   int i1, i2;
   int extra_data;
};

data_struct_improved* data_ptr;
data_ptr = new data_struct_improved;
//....
// Mistake
memset(data_ptr, '\0', sizeof(data_struct));
```

In the real world, there's going to be a lot of code between the declaration of `data_ptr` and the line clearing it:

```
memset(data_ptr, '\0', sizeof(data_struct));
```

So the programmer can be forgiven if he didn't see this line when he changed the type of the variable `data_ptr`. But and error has still been introduced and this is not good.

The Hack: Use `sizeof(*ptr)` to determine the size of dynamic data

No matter how you change the type of `data_ptr`, the expression `sizeof(*data_ptr)` will always contain the right number of bytes.

So this code always works:


```
memset(data_ptr, '\0', sizeof(*data_ptr));
```

Hack 34: Zero Out Pointers to Avoid Reuse

The Problem: Reusing pointers can corrupt memory

After a pointer is deleted (or freed) it should not longer be used. That doesn't prevent bad code from using it though. Sometimes this results in a random corruption of memory, one of the toughest problems to debug.

Here's an example of what were talking about:

```
int* data;

data = new int[10];
data[0] = 0;
// ... lots more work with data ...

delete[] data;

// ... a few thousand lines of code ...
data[2] = 2;   // Memory corruption
```

The Hack: Set the pointer to `NULL` after deleting it

For example:

```
delete[] data;
data = NULL;
```

The result is that on most systems if you try and use the pointer, the system will crash:

```
// ... a few thousand lines of code ...
data[2] = 2;   // System crashes
```

This hack does not prevent errors, but it does change them. Accessing a memory pointer after it is deleted can result in data corruption or heap corruption. The results of this error may not show up for a long time and will probably happen when you are executing a totally unrelated piece of code.

On the other hand, if the pointer has been set to `NULL`, then an attempt to use will usually result in a system crash as the point of use.

So by resetting pointers after each use you change a hard to find, random error, into one that repeatable and easy to figure out.

Hack 35: Use strncpy Instead of strcpy To Avoid Buffer Overflows

The Problem: The function `strcpy` is unsafe.

The following code overflows an array and corrupts memory.

```
// Bad code
char name[10];
strcpy(name, "Steve Oualline");
```

The Hack: Use `strncpy`

The function `strncpy` copies a string, but only a limited number of characters. If we add to this the `sizeof()` operator we can be assured that we will never copy more data than the destination can hold.

So a safer string copying method is:

```
// Partial solution
strncpy(name,  "Steve Oualline", sizeof(name));
```

But there's a catch. Reading the documentation for `strncpy` we find the following paragraph:

```
The strncpy() function is similar [to strcpy], except that not more than n bytes
of
src  are copied. Thus, if there is no null byte among the first n bytes
of src, the result will not be null-terminated.
```

We want the result to always be null terminated, so we explicitly handle the exception case and make sure that our string is null terminated. Again, we use the `sizeof()` operator to make sure we get the right size.

```
name[sizeof(name)-1] = '\0';
```

Putting this all together we get the code:

```
// Good code
char name[10];
strncpy(name, sizeof(name), "Steve Oualline");
name[sizeof(name)-1] = '\0';
```

Warning: The `sizeof(name)` only works if name is declared as an array. If it is declared as a pointer, then `sizeof(name_ptr)` returns the size of the pointer, not the data that it is pointing to.

Note: It is possible to avoid all C style string related memory problems by using C++ `std::string` strings. But a lot of code still uses the old style strings and there are performance issues with C++ strings.

Note: A new function called `strlcpy` has been introduced in some C/C++ libraries which allows for safe string copies. However, it's not standard, it's not in all libraries, and Solaris and OpenBSD have implemented it differently. But if you have it, by all means use it.

Hack 36: Use strncat instead of strcat for safety

The Problem: `strcat` is unsafe

The following code overflows the array name and corrupts memory.

```
// Bad code
char name[10];

strncpy(name, "Oualline", sizeof(name));
name[sizeof(name)-1] = '\0';

strcat(name, ", ");
strcat(name, "Oualline);
// Memory is now corrupt
```

Note: Defining an array using a numeric constant (`10`) instead of a named constant (`NAME_SIZE`) is bad programming practice. But it does make the example simpler, so it's good writing practice when explaining a hack.

The Hack: Always use `strncat`

Let's rewrite the previous example using safe programming hacks. First we safely copy "`Oualline`" into the `name` variable.

Next we use `strncat` to add on the <comma>, <space> characters. The question is how many characters can we put in the array? The variable `name` will hold up to 10 characters. But to be safe we spell 10 as `sizeof(name)`.

We've already used up `strlen(name)` characters, so the free space in the variable is denoted by the expression:

```
sizeof(name) – strlen(name)          // Incomplete
```

We need one byte for the end of string character. So let's add it to our expression:


```
sizeof(name) – strlen(name) -1
```

Putting it all together we get this safe version of the program:

```
// Good code
char name[10];

strncpy(name, "Oualline", sizeof(name));
name[sizeof(name)-1] = '\0';

// Concatenation example
strncat(name, ", ", sizeof(name) – strlen(name) -1);
name[sizeof(name)-1] = '\0';  // This is required

strncat(name, "Oualline, sizeof(name) – strlen(name) -1);
name[sizeof(name)-1] = '\0';
```

Note: A new function called `strlcat` has been introduced in some C/C++ libraries which allows for safe string catenation. However, it's not standard, it's not in all libraries, and Solaris and OpenBSD have implemented it differently. But if you have it, by all means use it.

Hack 37: Use snprintf To Create Strings

The Problem: When using **`sprintf`** it's possible to write data to the string that's longer than the string. For example:

```
char file[10];

sprintf(file, “/var/tmp/prog/session/%d”, pid);
```

The Hack: Use **`snprintf`** instead.

The second parameter to **`snprintf`** is the size of the string. In keeping with our other safety hacks, we use **`sizeof(`*`string`*`)`** in this parameter whenever possible. Since **`snprintf`** knows the size of the string, it is smart enough not to overflow it.

So the following code will not mess up memory. The file name will be a little short, but at least you won't have to deal with memory corruption.

```
char file[10];

snprintf(file, sizeof(file),
        “/var/tmp/prog/session/%d”, pid);
```


Hack 38: Don't Design in Artificial Limits

The Problem: Any time you design code with an artificial limit someone will exceed it. For example, consider the code:

```
void err(const char* const fmt, int a = 0, int b = 0,
         int c = 0, int d = 0, int e = 0)
{
    fprintf(stderr, "Fatal Error:\n");
    fprintf(stderr, fmt, a, b, c, d, e);
    fprintf(stderr, "\n");
    abort();
}
```

Now this works if you wish to write simple messages:

```
err(“Size parameter (%d) out of range”, size);
```

But what happens when we wish to a slightly more complex call?

```
err(“Point (%d,%d) outside of box (%d,%d), (%d,%d)”,
    point.x, point.y, box.x1, box.y1, box.x2, box.y2);
```

Our function can take a format and up to five parameters. We just gave it six. It's not going to work.

Now we could fix the `err` function to add another parameter, but that would only work until we needed seven parameter. Another change would be needed at eight and so on.

The Hack: When writing general functions don't design yourself into a corner. If you really know the C++ language then you know how to design an `err` function that takes any number that takes any number of parameters:

```
#include <cstdarg>

void err(const char* const fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    fprintf(stderr, "Fatal Error:\n");
    vfprintf(stderr, fmt, args);
    fprintf(stderr, "\n");
    abort();
}
```

This is not the only place where artificial limits can cause trouble. Any time you design limits into the system, sooner or later someone will exceed them.

One classic example of an arbitrary limitation was the 640K limitation. A quote attributed to Bill Gates[9] states "640K ought to be enough memory for anyone." But it was only a short while after the original PC came out that people started inventing add-on cards with special drivers designed to get around the 640K limitation.

Another example is the *ls* command. The initial version used single characters for options (**`-l`**, **`-R`**, etc.). Do a *man ls* now and you'll find that almost all the single letters (upper and lower case) are being used by this command. Fifty two (26*2) options were just not enough. By limiting options to a single character the designers of *ls* limited their expandability. Someone had to devise a new option syntax (**`--long-options`**) to get around this problem.

The Senior Citizen Truant

One state's citizen tracking system limited a person's age to two digits. So when one of its citizens reached 100 her age was reset to 0. This wasn't too big a problem.

The real trouble came when she reached 107 and the state sent a truant officer out to her house to find out why her parents had not enrolled her in the first grade.

Hack 39: Always Check for Self Assignment

The Problem: It is possible to create code that performs the following operation on a class:

```
x = x;
```

In real life self assignment is probably not going to be as obvious as this. Instead it will probably be hidden by a couple or more layers of function calls.

But when this does occur the results can be catastrophic. Consider the following assignment operator pseudo code:

```
a_class& opeator = (const a_class& other_one) {
   1. Delete my data
   2. Allocate a new structure the same size as
      other_one's data
   3. Copy over the data
```

The code for this class is:

```
class a_class {
```

9 Mr. Gates has denied saying this.


```
    private:
        unsigned int size;
        char* data;
    public:
        a_class(unsigned int i_size) {
           size = i_size;
           data = new char[size]);
       }

       // Here's the problem code
       a_class operator = (const& a_class other) {
            delete[] data;
            data = NULL;
            size = other.size;
            data = new char[size];
            memcpy(data, other.data, size);
       }
}
```

Now consider what happens when a variable is assigned to itself. The first step:

```
   1. Delete my data
```

deletes the data for the destination variable (`*this`). However, the destination variable is also the source variable (`other_one`), the data for this is deleted as well. In fact the only copy of the data is deleted.

The program will fail when we get to the step:

```
   2. Allocate a new structure the same size as
      other_one's data
```

because there is no data in `other_one`, we deleted it in step 1.

The Hack: Program defensively and check for self assignment explicitly.

Every non-trivial class should have the following at the beginning of each assignment operator function:

```
a_class& opeator = (const a_class& other_one) {
    if (this == &other_one)
        return (*this); // Self assignment detected
// ....
```

This little bit of insurance can prevent a really nasty problem from occurring.

Hack 40: Use Sentinels to Protect the Integrity of Your Classes

The Problem: Strange things are happening to one of your classes. Data is getting corrupted somehow. You suspect someone is writing over random memory, but how do you prove it:

The Hack: Put sentential at the beginning and end of your class and check them often.

For example:

```
class no_stomp {
    private:
        enum {START_MAGIC = 0x12345678,
              END_MAGIC = 0x87654321};
        // This must be the first constant or variable
        // declared.  It protects the class against
        // nasty code overwriting it.
        const long unsigned int start_sentinel;
// ....
    public:
        no_stomp(): start_sentinel(START_MAGIC),
                    end_sentinel(END_MAGIC)
       {}
// ....
       // Every member function should call this
       // to make sure that everything is still OK.
       check_sentinels() {
            assert(start_sentinel == START_MAGIC);
            assert(end_sentinel == END_MAGIC);
       }
// ....
   private:
       // This must be the last declaration
       const long unsigned int end_sentinel;
};
```

This class defines two constants which are declared at each end of the class. The term “constant” is slightly incorrect. They are really variables which are initialized when the class is created and can not be modified during the lifetime of the class.

But they can be overwritten if some buggy code overflows an array, uses a bad memory pointer, or does something else crazy. If that happens the next call to `check_sentinels` will cause an assertion failure crashing the program.

So what we are actually doing is reducing a difficult to solve problem (random overwrites of memory cause random results) to one that's easier to solve (as soon as the memory goes bad, the program crashes).

This type of code helped me locate a rather unusual problem with some C code what had been upgraded to C++. In this case the first call to `check_sentinels` crashed the program. Analyzing the program I discovered that the problem was caused somewhere between when the constructor was called and the first sentential verification call.

So I put a breakpoint in the constructor and was planning to single step through the code until I found the problem. The program crashed before the constructor was called. So I was left with a puzzle "How can a program call a member function before it calls the constructor?"

The answer was surprisingly simple. The class was being created with `malloc`. I told you it was once C. Turns out that it was not quite fully ported from C to C++. Replacing `malloc` with `new` solved the problem.

Hack 41: Solve Memory Problems with valgrind

The Problems: Bad pointers, writing of the end of allocated memory, memory leaks.

C++ gives you lots of flexibility when it comes to memory management. You are allowed to allocate and deallocate memory and directly manipulate pointers.

Flexibility comes with a cost. Because the language allows you to allocate memory, you can screw up the allocations. Similarly you can screw up the deallocation and use of pointers.

Since there are no built-in safety checks in C++ what do you do to protect your code?

The Hack: Use *valgrind*.

The *valgrind* program runs your program in a sort of virtual machine. Memory accesses undergo extra checking. That makes it possible to detect certain types of pointer errors. These include:

1. Using freed memory.
2. Writing past the end of a allocated block.

3. Writing past the beginning of an allocated block.
4. Memory Leaks

Let's take a look at a small example:

```
    1   #include <iostream>
    2
    3   int main()
    4   {
    5       int* ptr = new int[10];
    6       int* ptr_b = new int[10];
    7
    8       *(ptr+11) = 5;
    9       ptr_b = NULL;
   10       exit(0);
   11   }
```

Note: You may have noticed that we violated our own safety rules by not putting **`assert`** statements before each pointer access. But the purpose of this code is to cause problems, not catch them, so we disabled the safety's.

What are the problems with this code? On line 8 we access element 11 of a 10 element array. On line 9 we zero out the only pointer to the memory we allocated in line 6 thus causing a memory leak.

Let's look at what *valgrind* does when this program is run.

```
$ valgrind --leak-check=full ./bad_mem
==31755== Memcheck, a memory error detector.
==31755== Copyright (C) 2002-2006, and GNU GPL'd, by Julian Seward et al.
==31755== Using LibVEX rev 1658, a library for dynamic binary translation.
==31755== Copyright (C) 2004-2006, and GNU GPL'd, by OpenWorks LLP.
==31755== Using valgrind-3.2.1, a dynamic binary instrumentation framework.
==31755== Copyright (C) 2000-2006, and GNU GPL'd, by Julian Seward et al.
==31755== For more details, rerun with: -v
==31755==
==31755== Invalid write of size 4
==31755==    at 0x8048712: main (bad_mem.cpp:8)
==31755==  Address 0x4253054 is 4 bytes after a block of size 40 alloc'd
==31755==    at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755==    by 0x80486F5: main (bad_mem.cpp:5)
==31755==
==31755== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 21 from 1)
==31755== malloc/free: in use at exit: 80 bytes in 2 blocks.
==31755== malloc/free: 2 allocs, 0 frees, 80 bytes allocated.
==31755== For counts of detected errors, rerun with: -v
==31755== searching for pointers to 2 not-freed blocks.
==31755== checked 104,796 bytes.
==31755==
```



```
==31755==
==31755== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==31755==    at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755==    by 0x8048705: main (bad_mem.cpp:6)
==31755==
==31755== LEAK SUMMARY:
==31755==    definitely lost: 40 bytes in 1 blocks.
==31755==      possibly lost: 0 bytes in 0 blocks.
==31755==    still reachable: 40 bytes in 1 blocks.
==31755==         suppressed: 0 bytes in 0 blocks.
==31755== Reachable blocks (those to which a pointer was found) are not shown.
==31755== To see them, rerun with: --show-reachable=yes
```

After some chatter, the program catches the first thing the tool notices is the write to an illegal memory location:

```
==31755== Invalid write of size 4
==31755==    at 0x8048712: main (bad_mem.cpp:8)
==31755==  Address 0x4253054 is 4 bytes after a block of size 40 alloc'd
==31755==    at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755==    by 0x80486F5: main (bad_mem.cpp:5)
```

After the program finishes, *valgrind* checks the heap to see if any memory was lost. In this case it find that we allocated some memory at line 6 and lost it.

```
==31755== 40 bytes in 1 blocks are definitely lost in loss record 1 of 2
==31755==    at 0x4019D55: operator new[](unsigned) (vg_replace_malloc.c:195)
==31755==    by 0x8048705: main (bad_mem.cpp:6)
```

The tool is smart. It didn't report the memory we allocated on line 5 as lost even through we allocated it and never freed it. That's because at the time the program exited there was a pointer to this block of memory. In other words someone was using it at exit time, so it was not lost.

The *valgrind* tool is not perfect can not find problems with local or global arrays, only allocated memory. But still it does a very good job of finding a large number of difficult to locate problems.

Hack 42: Finding Uninitialized Variables

The Problem: Your program does strange things at random times. You suspect an initialized variable.

Consider the following example:

```
 1  #include <iostream>
 2
 3  static void print_state()
 4  {
 5      int state;
 6
 7      if (state) {
 8          std::cout << "State alpha" << std::endl;
 9      } else {
10          std::cout << "State beta" << std::endl;
11      }
12  }
```

At line 7 we use the value of `state` to decide if we do the alpha or beta parts. There's just one problem, we never bother to give state a value.

So which state will be executed? That depends on what the previous function left on the stack. In other words, `state` will be set to some random number.

How do you detect such things? It's difficult using the debugger. If you print the value of state and get 38120349, how do you know if that is correct or incorrect?

The Hack: *valgrind* to the rescue

The *valgrind* program also checks to see if you use any uninitialized data. It's actually rather clever about this. If you assign one uninitialized variable to another, it won't complain. (This actually happens a lot when doing `memcpy` and other similar operations.)

But if you try to use uninitialized data to make a decision (inside an **if** statement for example) it will complain loudly.

For example when we run the previous program under *valgrind* we get:

```
... chatter ...
==26488== Conditional jump or move depends on uninitialised value(s)
==26488==    at 0x80487DA: print_state() (uinit.cpp:7)
==26488==    by 0x804884C: main (uinit.cpp:16)
State alpha
... more chatter ...
```

This clearly shows us that line 7 has a problem with uninitialized data.

Hack 29: Valgrind Pronunciation

The Problem: How do you pronounce “Valgrind”?

The Hack: We hackers. We don't care how it's pronounced as long as it works.

Hack 43: Locating Pointer problems ElectricFence

The Problem: You need to find memory allocation errors.

The Hack: Don't use ElectricFence it's stagnant. Even its successor, DUMA is obsolete. Use *valgrind* instead.

Hack 44: Dealing with Complex Function and Pointer Declarations

The Problem: You need to create a pointer to a function which takes a integer argument and returns an array of pointers to functions which take one argument, a string, and return an integer.

Whew! That's a specification you have to read at least three times to figure out that you can't understand it.

How do you deal with such complexity.

The Hack: Redesign the code so you don't need this type of function. Let's assume that the world is insane and we can't do that. Then we need to break the declaration down into pieces and use lots of **typedef** statements to make things simpler. Hacking does not mean that you do the most complex thing in the most complex way. A good hacker does the most complex thing in a simple way.

Let's start by parsing the sentence and look for the bottom most element in this specification change. The base item is a function which takes one argument, a string, and returns an integer. It's easier said in C++:

```
typedef int base_function(const char* const arg);
```

Now we are going to return an array of pointers to this type of function. First we define a type for the pointer:

```
typedef base_function* base_function_ptr;
```

Next we define an array of pointers:

```
typedef base_function_ptr base_array[];
```


Now we need a function which returns a pointer to this array:

```
typedef base_array* big_fun(int value);
```

Finally we declare a pointer to this item:

```
big_fun* the_pointer;
```

Now you may have noticed that we used four **typedef** statements to define this one pointer. We could have done this in one statement.

But the goal of this hack was to define the pointer with a minimum of effort, not a minimum of code. Clarity is one quality highly valued by good hackers and by breaking down this problem into multiple, small **typedef** statement we can use many small clear statements in place of one large complex and impossible to understand C++ declaration.

The full code (including comments) follows:

```
// Type definition for a function which takes
// a character argument and returns an integer
typedef int base_function(const char* const arg);

// Pointer to a base function
typedef base_function* base_function_ptr;

// An array of pointers to our base function
typedef base_function_ptr base_array[];

// Function which returns an array of function
// pointers
typedef base_array* big_fun(int value);

// Finally a pointer to the thing we wished
// to point to
big_fun* the_pointer;
```

As hackers we are expected to think out of the box and look beyond the initial problem. One question that we haven't dealt with is “Why would you ever need such a pointer?”

In the real world, a good hacker would not design a better way to declare the pointer, he would design code so that a pointer like this was never needed in the first place. Elimination of needless complexity is one of the attributes of a truly great hacker.

Chapter 3: File Hacks

Hacking is not just about writing programs. Programs transform and process data. But the data itself can also be subject to hacking.

Designing a good file format is one key part of creating a good program. The files should be easy to use, robust, and expandable. The hacks contained in this chapter are designed to help you achieve all these goals.

The key to good file design is to make things as simple as possible, but no simpler, and as flexible as possible without getting silly. It also helps to be as transparent as possible so any problems with the file can easily be located.

As hackers we want our programs used as much as possible. This means designing file formats that are so good that people want to use them. XML is one example of a simple, yet robust file format. Simple enough so that anyone can read it (sort of) yet complex enough so you can specify just about anything.

Good programmer create good files. Good hackers create great ones.

Hack 45: Create Text Files Instead of Binary Ones Whenever Feasible

The Problem: You need save the configuration data (or other data) from your program. Do you use a binary file or and text file?

The Hack: Use text. It almost always make things easier.

UNIX is an excellent example of hacker design. One of the key design features that make the system work as well as it does is that almost all the configuration files are text.

There are numerous advantages to text files. The first is that they are human readable. When your program write a text file, the contents of the file can be examined by simpling looking at it. You don't have to create some fancy data dumper to look at the file.

This illustrates one of the major advantages of text files: transparency. You can see what's going on in a text file. Also you change change it easily using an editor. This means that if you program needs a configuration file and that file is in text format, then a text editor can be used to edit the configuration. (I didn't say it would be easy or pretty, but it can be done.)

Also if the configuration information text based, other people can easily write tools which edit the configuration file. On Linux you'll find literally hundreds of programs which configure network settings. That's because the network configuration is stored in a text file and easily accessible to everyone.

Again, one of your goals as a hacker is to make things a simple and easy to play with as possible. Text files let you do this.

Binary files are only useful when you are processing huge amount of data (video, graphics, databases, etc.) When you are dealing with 100,000,000 records or more, then the overhead associated with text files can make your program slow down tremendously. In that case binary files are better.

But for simple things like configuration files, settings files, and basic data text files are best. Performance is not an issue. Who cares if it takes 0.003 seconds to start your program instead of 0.001. But people do care about clarity, correctness, and interoperability and that where text files are best.

How Not to Design a Configuration System

The consequences of a bad design are worth study too. Let's look at one very common system for storing configuration settings.

First of all, all the configuration information for every program on the entire system is stored in one location. There are several problems with this:

1. A single wild program can accidentally totally destroy all the settings. (The solution to this problem: Don't redesign, the configuration system, provide a elaborate backup system so when destruction does occur you can recover – sort of.)

2. Make the file binary so no one can easily read it except with an API that you supply.

3. Keep the format secret and only the user edit the configuration data using the tool you provide. By keeping things secret you absolutely prevent anyone from writing a better program.

4. Because all settings are stored in one location, you've made it impossible to create several configuration files to be used for different situations. You limit everyone on the machine to one configuration period.

5. Finally because configuration data can only be stored on the local machine in a single location, configuration sharing is impossible.

The system I've been describing is the Microsoft Windows Registry. It is fascinating from a design point of view because it gives good designers some many opportunities to ask “What were they thinking when they did this?”

Hack 46: Use Magic Strings to Identify File Types

The Problem: Your program needs the user to specify a configuration file and a settings files. But the users can easily get the two mixed up causing havoc with your program.

The Hack: Use a magic string

Simply use a known string at the beginning of each type of file to identify it. For example, a typical configuration file might look like:

```
# TGP Configuration File
solve = true
features = novice
....
```

The key things to notice about the magic string are that it identifies the program it belongs to as well as the file type. So files that begin with this string and only files that begin with this string are configuration files.

Thus you prevent errors by forcing the user to give you a configuration file when a configuration file is needed.

Hack 47: Use Magic Numbers for Binary Files

The Problem: Our program uses binary files as input. How can we be sure we get the right binary files?

The Hack: Put a magic number at the beginning of the file.

Now the question is how do you decide on the magic number. The short answer is that you make it up.

There are many different ways of doing this. The method I prefer is to think up a word that describes the program I'm creating. For example “HACKS”. Taking the first 4 letters and seeing what the ASCII numbers for them I get:

H	48
A	41
C	43
K	4B

Putting these together gives us the number `0x4841434B`. The problem with this number is that because it is four ASCII characters and the file can easily be confused with a text file.

So we add `0x80808080` to the number giving us a `0xC8C1C3CB`. The result is our magic number:

```
// The magic number at the beginning of each hack file
const long unsigned int HACK_MAGIC = 0xC8C1C3CBL;
```

The final step is to write out a file with our new magic number in it and see if the Linux *file* command can identify it as an existing number.

XML Files

The XML file format has been designed to solve a great many of the problems with file formats. It's structured so that you can design quite complex things, it human readable (sort of), and you can easily validate it.

Hack 48: Automatic Byte Ordering Through Magic Numbers

The Problem: Binary files are not portable. Files written on a Sparc machine can not be read on a x86 machine.

The Hack: Use the magic number to detect file written with a different byte order and correct it.

For example:

```
// The magic number at the beginning of each hack file
const long unsigned int HACK_MAGIC = 0xC8C1C3CBL;

// The magic number on a machine with a different byte
// order
const long unsigned int HACK_MAGIC_SWITCH = 0xCBC3C1C8L;

// .....
long unsigned int magic;
in_file.read(&magic, sizeof(magic));

if (magic == HACK_MAGIC) {
    process_file();
} else if (magic == HACK_MAGIC_SWITCH) {
    process_file_after_flipping_bytes();
} else {
    throw(file_type_exception(
        "File is not a hack data file "));
}
```

The idea is simple, first check to see if we have a normal file and process it.

```
if (magic == HACK_MAGIC) {
    process_file();
```

We notice that a normal magic number is `0xC8C1C3CBL`. On a machine with a different byte order the magic number is `0xCBC3C1C8L`. If we detect a magic number that matches this value, we know we have a byte flipped file and process it:

```
const long unsigned int HACK_MAGIC_SWITCH = 0xCBC3C1C8L;
// ....
} else if (magic == HACK_MAGIC_SWITCH) {
    process_file_after_flipping_bytes();
```

Note: Although this system works, you do have to maintain two version of the file reading program. This can be a significant maintainable and risk problem. You may want to consider using the next hack instead.

Hack 49: Writing Portable Binary Files

The Problem: You need to create and read binary files which can be used on multiple machines.

The Hack: Always write out the data using "Network Byte Order".

In order to make data transmission platform independent a standard byte order called “Network Byte Order” was created. A number of functions were added to the C library to convert items to and from network byte order.

The include:

Function	***Meaning***
htonl	Convert a long in host format to network format.
htons	Convert a short in host format to network format.
ntohl	Convert a long in network format to host format.
ntohs	Convert a long in network format to host format.

Let's see how this might be used when writing a file:

```
// The magic number at the beginning of each hack file
const long unsigned int HACK_MAGIC = 0xC8C1C3CBL;

short int item_count = 15; // Number of items in the file
short int field_length = 11; // Length of next field

// ...
long unsigned int magic = htonl(HACK_MAGIC);
out_file.write(&magic, sizeof(magic));

short int write_item_count = htons(item_count);
out_file.write(&write_item_count,
               sizeof(write_item_count));

short int write_field_length = htons(field_length);
out_file.write(&write_field_length,
               sizeof(write_field_length));
```

Similar code is used on the read side to make things portable.

Hack 50: Make You Binary Files Extensible

The Problem: People keep wanting to extend programs by adding more features. If the program deals with binary files the file format must accommodate this.

The Hack: When designing files leave room for expansion. Use a file format which includes a record size in each record.

Here's a simple file format specification:

1. Record type (4 byte integer)

2. Record length (4 byte integer)

3. Record data (length – 8 bytes of data)

This is a deceptively simple format. Although simple, it leaves lots of room for expansion.

Let's start with the code used to read the records. It must read the first 8 bytes, get the length, then read the rest of the data. The entire record is then returned to the reader for processor. Very simple. A good hack.

All the program has to do is to look a the record type to determine what type of data is in the record. Then it can process the record.

Now what happens when we need to expand and enhance our data stream. All we have to do is to add a new record type. The reader doesn't have to be changed. It can still process the record because it doesn't have to know the type.

All we have to do is add a new record handler to the main program. For example:

```
struct record {
    int type;       // Record type
    int length;     // Length of record
    uint8 data[0];  // Data (variable length);
};

// ....
switch (record_var.type) {
    case REC_START:
        do_start(record_var);
        break;
    case REC_STOP:
        do_stop(record_var);
        break;
     case REC_PAUSE:
        do_pause(record_var);
        break;
     default:
         std::cout << “WARNING: Unknown record type “ <<
             record_var.type << “ Ignored” << std::endl;
         break;
}
```

With code like this it's easy to add a handler for a new record. New handlers easily fit into the schema.

Also this code will work for both past and future version of the file. It works with future versions because it skips records it does not know about. So if we add a new record type such as **REC_SKIP**, the program will still run. It won't process records it doesn't know about, but it won't crash either.

This built-in resilience is the hallmark of a really good hack.

Hack 51: Use magic numbers to protect binary file records

The Problem: Files can become corrupt. How do we protect ourselves against that.

The Hack: Put magic numbers at the beginning and end of each record. For example:

```
struct record_start {
    uint32 magic_start;
    uint32 type;
    uint32 size;
};

struct record_end {
    unit32 magic_end;
}
```

Somewhere in the recording code we have the statements:

```
if (record_stream.start.magic_start != START_MAGIC)
    throw(file_corruption("Starting magic number wrong");

// ....
if (record_stream.end.magic_end != END_MAGIC)
    throw(file_corruption("Ending magic number wrong");
```

Now when anyone monkeys with the file we notice. It should be pointed out that this protects against almost all simple data corruption caused by hardware errors or program bugs.

Also it only protects the ends of the data. If someone messes up the middle it won't be spotted. If you want to spot that you'll need to include a checksum of some sort in the data itself. But that's a level of paranoia that's rarely needed.

This hack has proved useful to myself in a number of occasions. The first was a file system problem. It seemed that when the computer was shutdown and there were files being written to the disk, the operating system would pad the unwritten sectors with blocks containing all zeros. Fortunately the magic numbers were there and the program realized that it had read something that was half a good record and half something else and discarded the data.

The other problem caught by this system was a really nasty data corruption bug. The problem was eventually traced to code in an entirely different module which sent out E-Mail alerts when an error condition occurred.

Here's the code. Let's see if you can spot the problem:

```
pid_t child_pid = fork();

if (child_pid == 0) {
    // We are the child
    system("send_email_alert");
    exit(0);
}
```

If you didn't spot the bug I can understand. It took me about a week of testing to pinpoint this code and locate the problem.

The problem is very hard to see. After all this entire module shares no code with the record writing module. In particular the output file handle does not exist at all outside the record writer.

Also the code in question does no I/O. It doesn't even manipulate memory so it couldn't be memory corruption. So what is going on?

There are two parts to this problem. The first is the call:

```
pid_t child_pid = fork();
```

This creates a duplicate of the main process. All file handles are now shared between the two processes. So where before there was one process with the output file open, now there are two.

The `fork` call also duplicates the memory of the parent process. This includes all the I/O buffers for all the files. This includes the output file.

Next we come to the line:

```
    exit(0);
```

All this does is exit the program, right? Not exactly. Let's take a look at the documentation for this function:


```
The exit() function causes normal program
termination and the the value of status & 0377 is
returned to the parent (see  wait(2)).  All
functions registered with atexit() and on_exit()
are called in the reverse order of their
registration, and all open streams are flushed
and closed.  Files created by tmpfile() are
removed.
```

The **exit** function actually does quite a lot. In particular it flushes the write buffers for any files open for writing. Since we inherited a set of open files and their buffers from the parent, they will be flushed. Thus this call writes some data to the file. The main program will also write some data to the file when it's buffers get full. Since these two writes are not coordinated the data is corrupted.

This was a nasty problem to find. But the fact that the code was written with record protection in it, catching the problem was much easier. Early on it was obvious that turning on alerts cause the program to complain about record corruption. The only hard part was looking at the code and trying to figure out the problem.

What made this nasty is that the logs were getting corrupted, but only if alerts were turned on. But these modules had nothing in common. The shared no functions, logic, or memory. It took quite some time to find out that the problem was that they did share something: I/O buffers, but that sharing was very well hidden.

Hack 52: Know When to Use _exit

The Problem: You are forking off a process and do some work and then exit, without doing anything to the I/O buffers in the process. You can't use the **exit** function because of the problems described in Hack 51.

The Hack: Use **_exit**.

The **_exit** function stops your program. That's all it does, stop the program. It does not pass go, it does not collect $200. But more importantly it does not flush any I/O buffers. All that happens when you call **_exit** is that your process goes away.

This is very useful when you've forked of a process that has done it's work and needs to stop. Remember **exit** flushes things and can cause all sorts of trouble with shared files. The **_exit** call avoids this problem.

Chapter 4: Debugging Hacks

I have written two programs which had no bugs. One was three machine instructions long and the other was four. All the others had bugs.

The writing of a program takes only a short time. Debugging goes on forever. The debugging hacks presented here are designed to make debugging faster and more efficient.

Hack 53: Mark temporary debugging messages with a special set of characters.

The Problem: Debugging by adding print statements is still a very powerful debugging technique. But how do you identify print statements designed to output information for a specific problem vs. the print statements that should be there.

The Hack: Begin all temporary debugging output with the characters "##". For example:

```
i = find_index();
std::cout << "## find_index returned " << i << std::endl;
```

The "##" serves several purposes. First, it identifies the statement as a temporary debug statement. Next when you do find the problem it's easy to remove them. All you have to do is go through and find each line containing a "##" and delete it.

Hack 54: Use the Editor to Analyze Log Output

The Problem: When you turn on verbose logging you get 50,000 lines scrolling past my screen and I can't find what I'm looking for.

The Hack: Save the log output to a file and use a text editor to browse the output.

The problem with log files is that they give you too little information or too much. Too little is easily dealt with – all you have to do is turn up the verbosity level until you get too much or put in ## lines. (See previous hack.)

Dealing with too much is another problem. How do you find that nugget of information which tells you just want you need to know?

Well on most systems there's a tool designed for the searching and manipulation of large text files. It's called your system editor. Let's see how this works in action.

First let's suppose you've done an exhaustive test of your system and logged the results. Out of 5,000 tests, three fail with an error. Simple start up your editor and search for the string “ERROR”. You'll locate the line of the first error.

Want to see what happened just before the error, scroll up and take a look.

The editor not only lets you look at the data but annotate it as well. You can add comments and annotations to the log as you figure out what is going on. And unlike paper notes, edits in the file can be inserted into an E-Mail in case you have to use the Internet for help.

Log files are a great source of information and the text editor is a great way to exploit this information.

(See Hack 127 for information on how to *Vim* to examine log files.)

Hack 55: Flexible Logging

The Problem: When your dealing with small programs it's OK to log everything. If you have a larger program you need to be more selective.

The Hack: Use letter based logging selection.

We wish to create a command line parameter `-v<letters>` which will enable debugging for only those sections specified by `<letters>`. For example a typical set of letters might be:

m Log memory allocation and frees.

d Log dictionary entries

f Print function definitions

r Print regular expression debugging information

x Display calls to the execute function

... and so on.

This system is surprising easy to implement. First we define an array to hold our debug options:


```
static bool verbose_letters[256] = {0};
```

Next we create a loop to process the command line arguments. In this example we're are using the GNU **`getopt`** function to scan the arguments. We use the argument specification `"v::"` to indicate the only option is `-v` and that it can be followed by options parameters. (This is indicated by the "::" after the "v".)

```
while (1) {
    int opt = getopt(argc, argv, "v::");
```

Now we process the options. There are two possible ways of specifying -v. The first is just -v alone. In that case we turn on all debugging information with the line:

```
      memset(verbose_letters, -1,
             sizeof(verbose_letters));
```

But if the options has arguments then we only set the letters of the given options:

```
    for (unsigned int i = 0; optarg[i] != '\0'; ++i)
       verbose_letters[
          static_cast<int>(optarg[i])] = true;
```

Putting it all together we get the following code for parsing the verbose option:

```
switch (opt) {
  case 'v':
    if (optarg == 0) {
      memset(verbose_letters, -1,
              sizeof(verbose_letters));
    } else {
     for (unsigned int i = 0; optarg[i] != '\0'; ++i)
       verbose_letters[static_cast<int>(optarg[i])] =
                        true;
    }
    break;
// ... process other options
```

Checking to see need to issue a debugging message is simple. For example to check for `malloc` logging we use the code:

```
if (verbose_letters['m']) {
    std::cerr << "Doing a malloc of " << size << bytes;
}
ptr = malloc(size);
```


In actual practice we would define a constant for `'m'`, but for this short example we'll excuse the bad programming practice.

This way of specifying what to log allows us to use all the lower case letters, all the upper case letters, and all the digits to select what to log. And if that's not enough there are lost of punctuation characters we can use as well.

(And if that's not enough, your program is probably too big to ever be debugged anyway.)

Two attributes make this a good hack. First it is simple. Second it is extremely flexible. In other words it a simple solution to a complex problem, and simple solutions are always good.

Hack 56: Turn Debugging On and Off With a Signal

The Problem: You want debugging output some of the time but not all of the time.

The Hack: Use a signal to turn on and off debugging.

The first step is to define a signal hander that toggles the debug flag:

```
static void toggle_debug(int) {
    debug = ! debug;
}
```

Next we connect it to a signal, in this case **SIGUSR1**.

```
    signal(SIGUSR1, toggle_debug);
```

Now all we have to do is make the logging conditional on the **debug** flag.

```
static void log_msg(const char* const msg) {
    if (debug) {
        std::cout << msg << std::endl;
    }
}
```

Now all we have to do to turn on debugging is use the command:

```
$ kill -USR1 pid
```

Turning it off is the same thing.

This hack is useful when you must turn on debugging output at a specific time. Normally it's easier to just turn it on when you first start the program and then leave it on.

But sometimes you need to instrument the main loop of a program that mysteriously hangs after four days of operations. Since you don't have a couple of terabytes of free disk space for the log files, you need to turn on debugging after the program hangs in order to find out what is going on that is making things nuts. That's where this hack is most useful.

Hack 57: Use a Signal File to Turn On and Off Debugging

The Problem: You need to turn on and off debugging and signals are just not practical.

The Hack: Use a special file to trigger debugging.

For example:

```
static void log_msg(const char* const msg) {
    if (access("/tmp/prog.debug.on", F_OK) != 0) {
        std::cout << msg << std::endl;
    }
}
```

Now all you have to do to turn on debugging is to create the file */tmp/prog.debug.on*. To turn it off simply remove the file.

This hack should be only used when you absolutely need to turn debugging on and off while the program is running and you can use signals (Hack 56). The access system call is expensive and will slow your program down if used frequently. So although this hack is useful, be aware of its limitations.

Hack 58: Starting the Debugger Automatically Upon Error

The Problem: Your program has detected an internal problem that needs debugging. But the program is not being debugged. How do you let the programmer at the problem?

The Hack: Define a function which starts the debugger on a running program.

Note: The following code is Linux specific. If you are running on a UNIX like system it should be easy to port it to that system. If you are running Microsoft Windows, you're on your own.

The basic idea of the program is that when a `debug_me` function call occurs that the program should start the debug (*gdb*) and attach it to the running process. Sounds simple, but there are a few details to work out.

First let's see what we need to tell *gdb* start the debugging process. The initial *gdb* commands are:

1. `attach <pid>` – Attach the debugger to the program being debugged.
2. `echo "Debugger gdb started\n"` – Let the user know what has happened.
3. `symbol /proc/<pid>/exe` – Tell *gdb* where to find the symbol table for the process.
4. `break gdb_stop` – Stop at a nice stopping point.
5. `shell touch <flag-file>` – Tell the program that *gdb* is attached. (More on this later)
6. `continue` – Continue execution and stop at the correct location.

The first *gdb* command:

```
attach <pid>
```

attaches the debugger to the program. (***<pid>*** is replaced by the process id of the program to be debugged.) The debugger is now in control of the program. Actually if we were in a minimalist frame of mind, we would stop here.

But the debug session is in a sorry state. The symbol table has not been loaded and don't know if we stopped in the correct thread or at a know location. So we execute a few more commands to make things a little nicer.

The next command:

```
echo "Debugger gdb started\n"
```

outputs a greeting message. That way the user that we've started the debugging process.

Next we load the symbol table. For that we need the name of the program file. One way of finding this is to talk look at the program name and do a PATH search for the executable file. But Linux is nice to provide a symbolic link from `/proc/<pid>/exe` to the executable, so we just exploit this feature to load our symbol table.


```
symbol /proc/<pid>/exe
```

Now we tell *gdb* to stop at a good place. In fact we've defined a good place to stop called `gdb_stop`, so we'll set a breakpoint there.

```
break gdb_stop
```

When the debugger is attached to a program, the program stops. The problem is we don't know where the program is stopped. It could be 80 levels deep into some function called by `debug_me`. What worse, we could be dealing with a threaded program. In that case we many not even be stopped in the thread that caused the error.

The solution to this problem is to set a stop in a know location (`gdb_stop`) and tell the debugger to continue. When we stop at `gdb_stop` we know where we are and we are sure to be in the correct thread.

After `debug_me` starts *gdb* it waits around for the debugger to start. This is done using the loop:

```
96:     while (access(flag_file, F_OK) != 0)
97:     {
98:         sleep (1);
99:     }
```

This loop waits around for a flag file to be created. As soon as it shows, the program knows that the debugger is running and it can continue.

In order to create the flag file, we issue the following command to *gdb:*

```
touch <flag_file>
```

Finally we tell *gdb* to continue. At this point the execution of the program continues for a short while until the program reaches `gdb_stop`.

The code to do all this work is listed in the full *debug_me.c* module at the end of this hack. Mostly it's a matter string processing to get the commands into the command file.

Now let's talk about the actual invocation of the *gdb* command. Ideally we should be able to just use a system call to execute the command:

```
gdb --command=<command-file>
```

In this example *<command-file>* will be replace by a temporary file containing the commands we listed above. But it's not as simple as that. It never is.

What is our program is a daemon running in background. It has no standard in and standard out. If we started *gdb* there would be no terminal in which to type commands.

A solution to this problem is to start our own terminal window. This done with the command:

```
xterm -bg red -e gdb –command=<command-file>
```

This starts a new *xterm* program in which our debugger will run. We set the background to red using the options `-bg red`. Red is used because it gets our attention. Besides the red screen of death sounds better than the blue screen of death.

Finally we tell *xterm* to execute the *gdb* command through the use of the `-e` option.

Now that we've done all this let's see how this function might be used in a program. Here's some code that handles a variable that it either black or white (at least under normal, sane circumstances):

```
#include “debug_me.h”
// ....
switch (black_or_white) {
    case BLACK:
        do_black();
        break;
    case WHITE:
        do_white();
        break;
    default:
        std::cerr << “INTERNAL ERROR: Impossible color” <<
            std::endl;
        debug_me();
        break;
}
```

In this case the variable `black_or_white` undergoes a sanity test. If things are insane we start the debugger.

Note: This only works for programs which are used internally. If you are giving a program to a customer without source code, this system is not that useful.

The full source code for the *debug_me.c* file follows.


```
 1: /*********************************************
 2:  * debug_me -- A module to start the debugger   *
 3:  *      from a running program.                  *
 4:  *                                               *
 5:  * Warning: This code is Linux specific.        *
 6:  *********************************************/
 7: #include <stdio.h>
 8: #include <unistd.h>
 9: #include <sys/param.h>
10: #include <stdlib.h>
11:
12: #include "debug_me.h"
13:
14: static int in_gdb = 0;    // True if gdb started
15:
16: /*********************************************
17:  * gdb_stop -- A place to stop the debugger     *
18:  *                                              *
19:  * Note: This is not static so that the         *
20:  * debugger can easily find it.                 *
21:  *********************************************/
22: void gdb_stop(void)
23: {
24:     printf("Gdb stop\n");fflush(stdout);
25: }
26:
27: /*********************************************
28:  * start_debugger -- Actually start             *
29:  *               the debugger                   *
30:  *********************************************/
31: static void start_debugger(void)
32: {
33:     int pid = getpid();         // Our PID
34:
35:     // The name of the gdb file
36:     char gdb_file_name[MAXPATHLEN];
37:
38:     // File that's used as a flag
39:     // to signal that gdb is running
40:     char flag_file[MAXPATHLEN];
41:
42:     // The file with the gdb information in it
43:     FILE *gdb_file;
44:
45:     // Command to start xterm
```



```
46:     char cmd[MAXPATHLEN+1OO];
47:
48:     if (in_gdb)
49:         return; /* Prevent double debugs */
5O:
51:     /*
52:      * Create a command file that contains
53:      *    attach <pid>  # Attaches to the process
54:      *    echo ....     # Echos a welcome message
55:      *    symbol /proc/<pid>/exe
56:      *                  # Loads the symbol table
57:      *    break gdb_stop # Set a breakpoint in
58:      *    shell touch /tmp/gdb.flag.<pid>
59:      *             # Create a file that tells us
6O:      *             # that the debugger is running
61:      *    continue # Continue the program
62:      */
63:     sprintf(gdb_file_name, "/tmp/gdb.%d", pid);
64:     gdb_file = fopen(gdb_file_name, "w");
65:     if (gdb_file == NULL)
66:     {
67:         fprintf(stderr,
68:                 "ERROR: Unable to open %s\n",
69:                 gdb_file_name);
7O:         abort();
71:     }
72:     sprintf(flag_file, "/tmp/gdb.flag.%d", pid);
73:     fprintf(gdb_file, "attach %d\n", pid);
74:     fprintf(gdb_file, "echo "
75:             "\"Debugger gdb started\\n\"\n");
76:
77:     fprintf(gdb_file, "symbol /proc/%d/exe\n",
78:             pid);
79:
8O:     fprintf(gdb_file, "break gdb_stop\n");
81:
82:     fprintf(gdb_file, "shell touch %s\n",
83:             flag_file);
84:
85:     fprintf(gdb_file, "continue\n");
86:     fclose(gdb_file);
87:     /* Start a xterm window with the
88:      * debugger in it */
89:     sprintf(cmd, "xterm -fg red "
9O:             "-e gdb --command=%s &",
```



```
91:             gdb_file_name);
92:     system(cmd);
93:
94:     /* Now sleep until the debugger starts and
95:      * creates the flag file */
96:     while (access(flag_file, F_OK) != 0)
97:     {
98:         sleep (1);
99:     }
100:     in_gdb = 1;
101:     gdb_stop();
102: }
103:
104: /*************************************************
105:  * debug_me -- Start the debugger                *
106:  *************************************************/
107: void debug_me(void)
108: {
109:     start_debugger();
110:     gdb_stop();
111: }
```

Hack 59: Making assert Failures Start the Debugger

The Problem: A failed `assert` normally just prints a message and aborts the program. This is not very useful when it comes to finding problems. It would be much better if a failed `assert` could start the debugger.

The Hack: Marry assertion failure with `debug_me`.

The first thing we need to do is to determine the name of the function that's called if an assertion fails. This varies from system to system.

To find out how assert works, we create a small test program:

```
#include <cassert>

assert("We are failing" != 0);
```

Next we run the program through the preprocessor and take a look at the output. Here's a example using the *gcc -E* command. The interesting lines are:

```
# 2 "assert.cpp" 2
```



```
(static_cast<void> (__builtin_expect (!!("We are
failing" != 0), 1) ? 0 : (__assert_fail ("\"We are
failing\" != 0", "assert.cpp", 3, __PRETTY_FUNCTION__),
0)));
```

From this we can see that the function `__assert_fail` is called when an assertion is triggered. Earlier in the output the compiler is even nice enough to provide us with a prototype:

```
extern “C” {
    extern void __assert_fail (
        __const char *__assertion,
        __const char *__file,
        unsigned int __line,
        __const char *__function)
    throw () __attribute__ ((__noreturn__));
}
```

Now all we have to do is supply our own version of `__assert_fail` which calls `debug_me`.

```
/*************************************************
 * __assert_fail -- Called when an assert fails.  *
 *      Starts the debugger.                      *
 *                                                *
 * Note: gcc specific.  Different compilers use   *
 *      different internal routines to handle bad *
 *      asserts.                                  *
 *************************************************/
void __assert_fail(
    const char *const what,
    const char *const file,
    const int line,
    const char *const funct
)
{
    printf("Assert failed: %s\n", what);
    printf("FAILURE at: %s:%d\n", file, line);
    printf("Function is %s\n", funct);

    debug_me();
    abort();
}
```

One thing to notice about this function is that it calls `abort` after `debug_me`. That's to prevent a careless programmer from typing continue in a debugging session and attempting to continue a obviously failed program.

The History of debug_me

The `debug_me` function was originally written to debug a mobile phone simulation program. The program had been abandon for a long time because it was extremely buggy. When I got it the program was extremely buggy and extremely out of date.

The program made extensive use of threads and frequently a thread would crash. Finding out which thread had cause the problem was extremely difficult as the *gdb* thread related commands didn't work too well on this system. So I invented `debug_me` to help solve this problem.

One interesting aspect of this program was its unusual failure modes. Quite frequently a thread would trigger an assertion failure and start the debugger. While the debugger was staring, another thread would fail and attempt to start the debugger.

Any programmer can write code that causes an assertion to fail and abort the program. It takes a real hacker to get multiple assertions to fail in a single run.

Hack 60: Stopping the Program at the Right Place

The Problem: You know that the program fails during the 487 iteration of the loop. If you put in a breakpoint you'll have to type continue 486 times before you get to the problem.

The Hack: Create a debugging point.

Some debuggers make it difficult to stop the program when you want to, like the 487th pass through a loop. So how get you get around this limtiation?

By putting in some temporary code just for the debugger. For example:

```
#ifndef QQQ
void debug_stop() {
    // The debugger can stop here
}
#endif /* QQQ */

// ...
```



```
for (i = 0; i < 5000; ++i) {
#ifndef QQQ
    if (i == 487) debug_stop();
#endif /* QQQ */
    // Problem code follows
```

Now all we have to do is to start our debugger and put a breakpoint in `debug_stop`. Right before the problem occurs this procedure will be called throwing us into the debugger. We can then single step through the program until we find the error.

There are a couple of things to note about this code. First debug_stop is a global function. The reason for this is that some debuggers have a hard time finding static functions.

The reason we use `#ifdef QQQ` for our debugging code is described in Hack 72. Basically it's an easy to way to identify temporary code.

Chapter 5: Commenting and Navigation Hacks

When I studying computer science at the university, the next great thing that was coming was the technology to typeset programs and break out of the monospaced font that we then used for programming. That was thirty years ago.

Today we still program using monspaced ASCII characters. No typesetting, no graphics – nothing that wasn't available thirty years ago. (With the possible exception of syntax highlighting.)

However hackers have never let inadequate technology stand in our way. In this chapter we present some of the hacks that let us get around the limitations of our programming environment.

Hack 61: Creating Headings within Comment

The Problem: The file format doesn't support headings, so how do you set your headings apart.

The Hacks: There are a number of tricks that you can play with your comments. First headings can be underlined or double underlined.

```
/*
 * A Important Heading
 * -------------------
 *
 * Here we put a paragraph discussing the stuff we've
 * headlined above.
 *
 * A Very Important Heading
 * ========================
 *
 * Headings like the one above can be used to denote
 * very important sections.
 *
 *  +-------------------------+
 *  | This is almost shouting |
 *  +-------------------------+
 *
 * Now here's something that really stands out.
```

There are other ways of indicating a major section break. For example you can make a heading that spans the entire line:

```
/*
```



```
 * <<<<<<<<<<<<<<<< I/O Section Follows >>>>>>>>>>>>>>>>
 */
```

And finally we have a really important section header:

```
/*************************************************************
 ********************** WARNING **************************
 ********************** WARNING **************************
 ********************** WARNING **************************
 *************************************************************/
/*
 * Do not attempt to drive heavy equipment while reading
 * this book.
 */
```

Hack 62: Emphasizing words within a paragraph

The Problem: The above works for headings, but what do you do when you want to typeset words within a paragraph.

The Hack: Creative use of “*” and other characters can help emphasize words.

For example:

```
/*
 * Some people consider all capitals to be SHOUTING!
 *
 * But there are other ways to **emphasize** a word.
 * More that <<one>> in fact.  After all there are a
 * number of ==fun== characters you can play around with.
 */
```

Hack 63: Putting Drawings In Comments

The Problem: There are no drawing functions in program editors.

It's impossible to put figures and graphs inside comments.

The Hack: Use ASCII art.

Primitive figures can be drawn using just the ASCII characters.[10] For example, the following comment documents the layout of a GUI:

10 Actually some very sophisticated ASCII art can be produced. Just not by programmers.


```
/*
 *   +-----------------------------------------+
 *   |  Name:          _____________________   |
 *   |  Address:       _____________________   |
 *   |  City:          _____________________   |
 *   +-----------------------------------------+
 *   |<- NAME_SIZE ->|                         |
 *                   |<--- BLANK_SIZE ---->|
 */
```

In this example we've not only created a figure but at the same time documented the constants we used for the dimension constants `NAME_SIZE` and `BLANK_SIZE`.

Hack 64: Providing User Documentation

The Problem: You need to provide user documentation for your program.

Oualline's law of documentation states: 90% of the time the documentation is lost. Out of the remaining 10%, 9% of the time it will be so out of data as to be totally useless. The remaining %1 of the time you have the correct documentation and the correct revision of the documentation it will be written in Japanese.[11]

The Hack: Use Perl's POD documentation system.

POD (Plain Old Documentation) format is a simple, yet fairly robust way of embedding documentation in a program. You can easily put a POD documentation section at the beginning of your program.

For example:

```
/*
 * The following is the documentation for the program.
 * Use the
=pod

=head1 NAME

solve_it - Solve the worlds problems

=head1 SYNOPSIS
```

11 The first I told this joke to a colleague he laughed for about three minutes then handed me his copy of Hitachi *Fortran Volume II. (Volume I was at the translator's.)*


```
    solve_it [-u] problem

=head1 DESCRIPTION

The I<solve_it> program solves all the worlds problem.
Just input the problem on the
command line and the solution will be output.

=head1 OPTIONS

The I<solve_it> takes the following options:

=over 4

=item B<-u>

Solve the problem for the universe, not just the world.

=back

=head1 LIMITATIONS

Currently totally unimplemented.

=head1 AUTHOR

Steve Oualline, E<lt>oualline@www.oualline.comE<gt>.

=head1 COPYRIGHT

Copyright 2007 Steve Oualline.
This program is distributed under the GPL.

=cut
*/
```

Now you can run your source code through one of the POD converters such as *pod2text* and get a formatted version of your documentation:

```
NAME
    solve_it - Solve the worlds problems

SYNOPSIS
       solve_it [-u] problem
```



```
DESCRIPTION
    The *solve_it* program solves all the worlds problem. Just input the
    problem on the command line and the solution will be output.

OPTIONS
    The *solve_it* takes the following options:

    -u  Solve the problem for the universe, not just the world.

LIMITATIONS
    Currently totally unimplemented.

AUTHOR
    Steve Oualline, <oualline@www.oualline.com>.

COPYRIGHT
    Copyright 2007 Steve Oualline. This program is distributed under the
    GPL.
```

Other converters include *pod2html*, *pod2man*, as well as others.

Hack 65: Documenting the API

The Problem: How do you document the API to a library you've written in a nice way.

Oualline's Law of Documentation applies to API documentation as well as the user manual.

The Hack: Use *doxygen* to automatically generate documentation from special comments in the code.

This tool allows you to use specially formatted comments to in-line documentation inside your code. Here's a short example:

```
/**
 * \brief The basic report specification
 *
 * This class completely describes all the parameters
 * needed to produce a report
 */
class ReportSpec {
    /** Name of the report */
    public:
        const char* const name;

    /**
     * Add a new report parameter
     *
     * \param name Name of the parameter
     * \param value Value of the parameter
     */
    void add_param(
            const char* const name,
            const char* const value
    );
};
```

When run through *Doxygen* this produces set of HTML documentation pages which can be viewed in any browser:

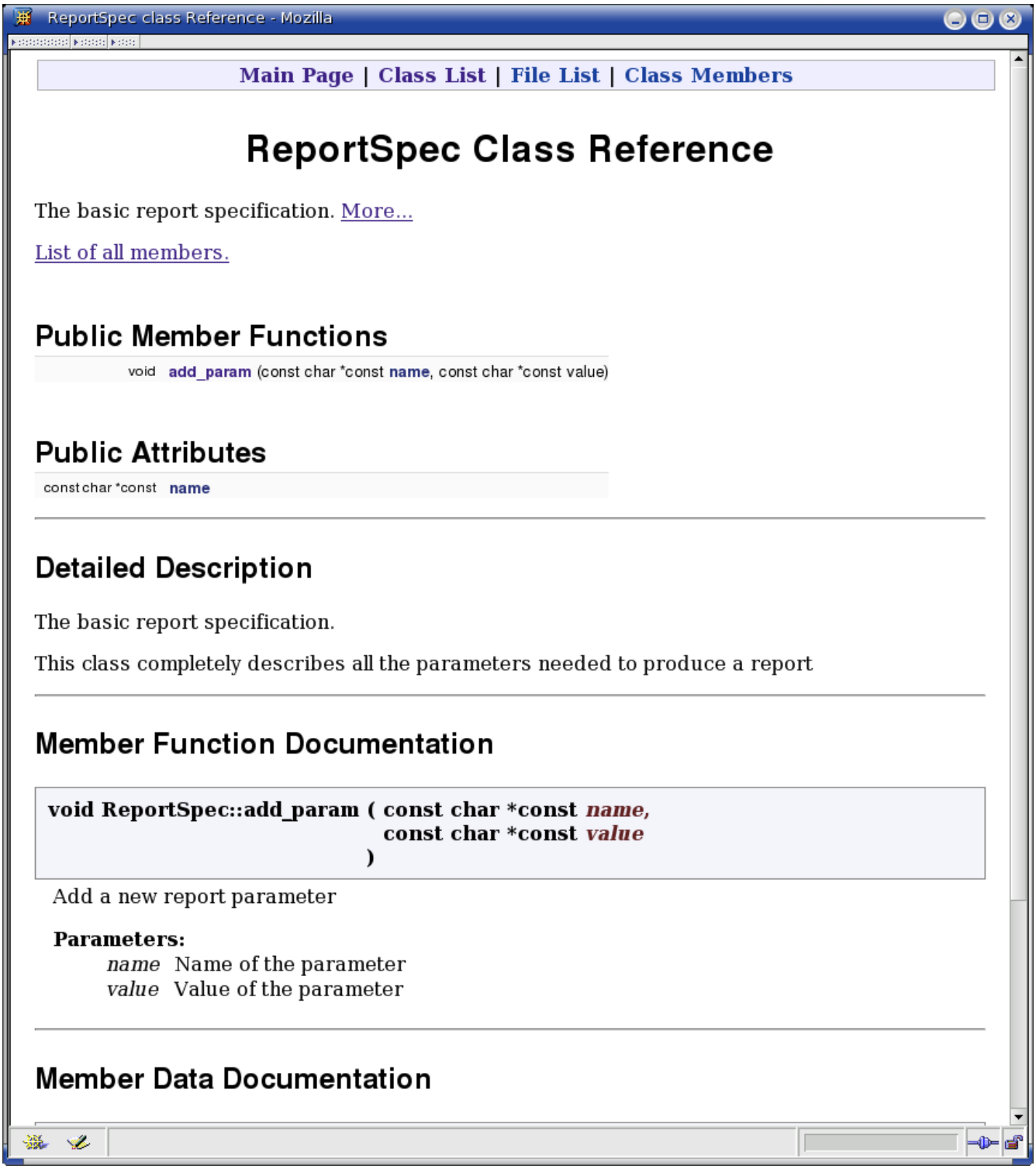


Figure 1: Doxygen documentation as HTML

This system does provide an excellent system for documenting your programs if you're willing to follow the conventions and write the comments correctly. On the other hand if you're dealing with a lot of code that didn't do this, check out the next hack.

Hack 66: Use the Linux Cross Reference to Navigate Large Coding Projects

The Problem: You're dealing with a poorly designed 5,000,000 line project and you're getting lost.

The Hack: Use the Linux Cross Reference to generate a cross referenced version of your code.

The Linux Cross Reference (lxr)[12] is a program which produces a hyper-text based markup of your code. You can get a copy from http://lxr.linux.no/.

It has a number of nice features such as the ability to locate any identifier in the code and show every place that it is used. Figure 2 shows the system being used to browse a file.

12 Don't let the name fool you. It will cross reference things other than Linux.

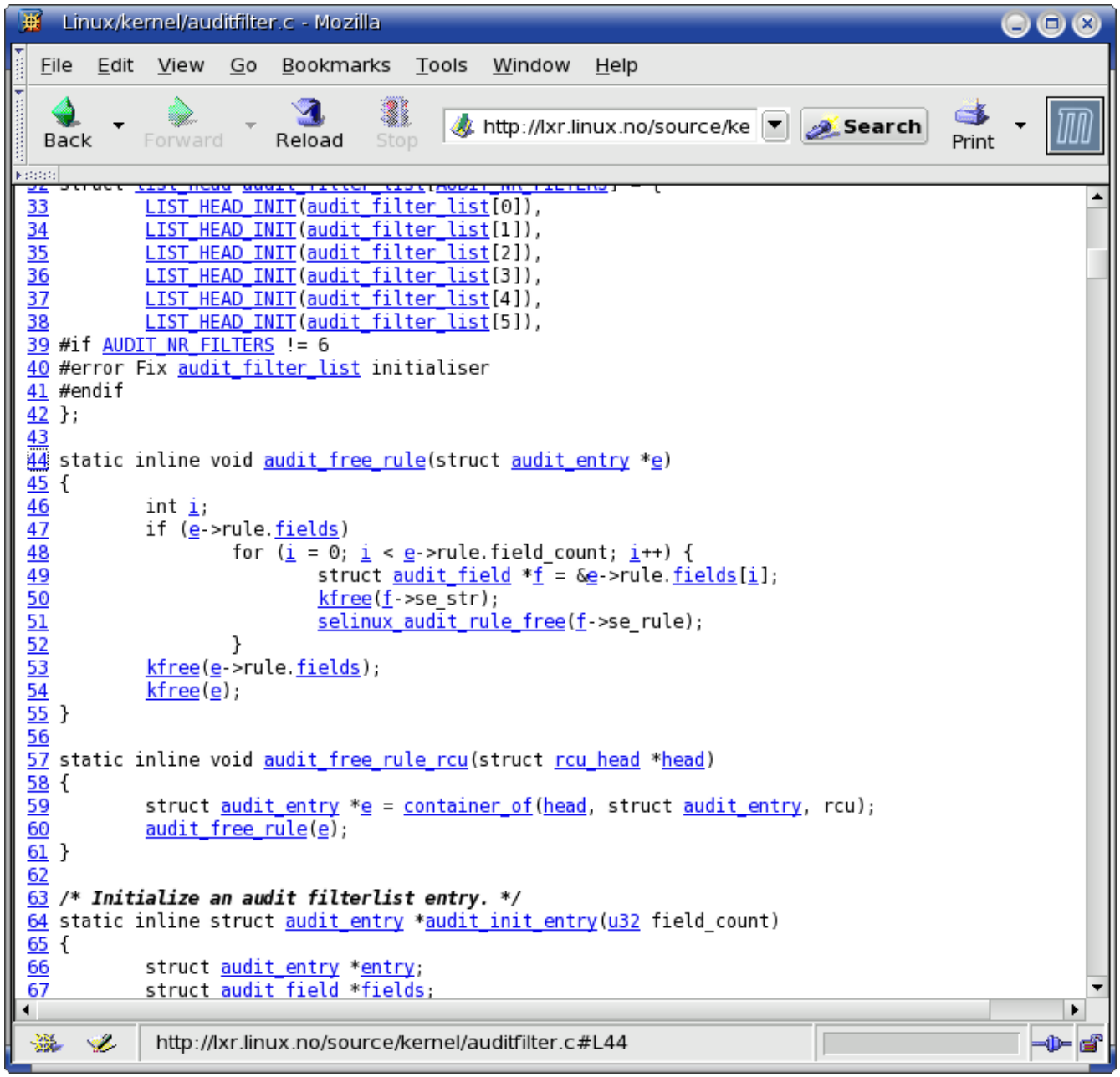


Figure 2: Linux Cross Reference (File View)

On line 44 there is a function that uses the `audit_entry` structure. We would like to know what an `audit_entry` is. Clicking on the identify takes to a page which lists every place that the identifier is used. (See Figure 3.)

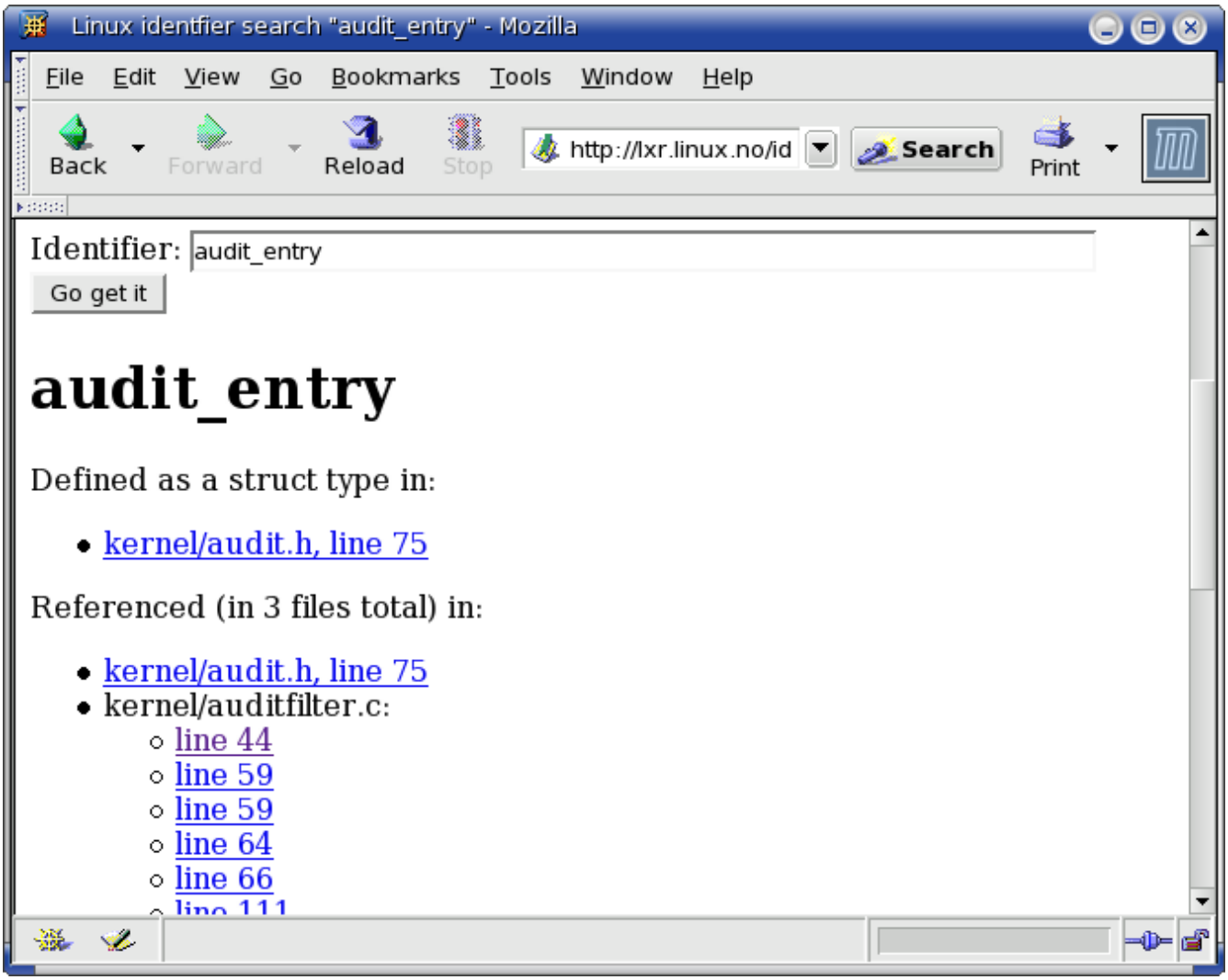


Figure 3: Identifier Use Page

This shows use that the identifier is defined in the file *kernel/audit.h* (line 75) and used in the file *kernel/auditfilter.c* in lots of places. Clicking on the first entry gives us the definition as shown in Figure 4

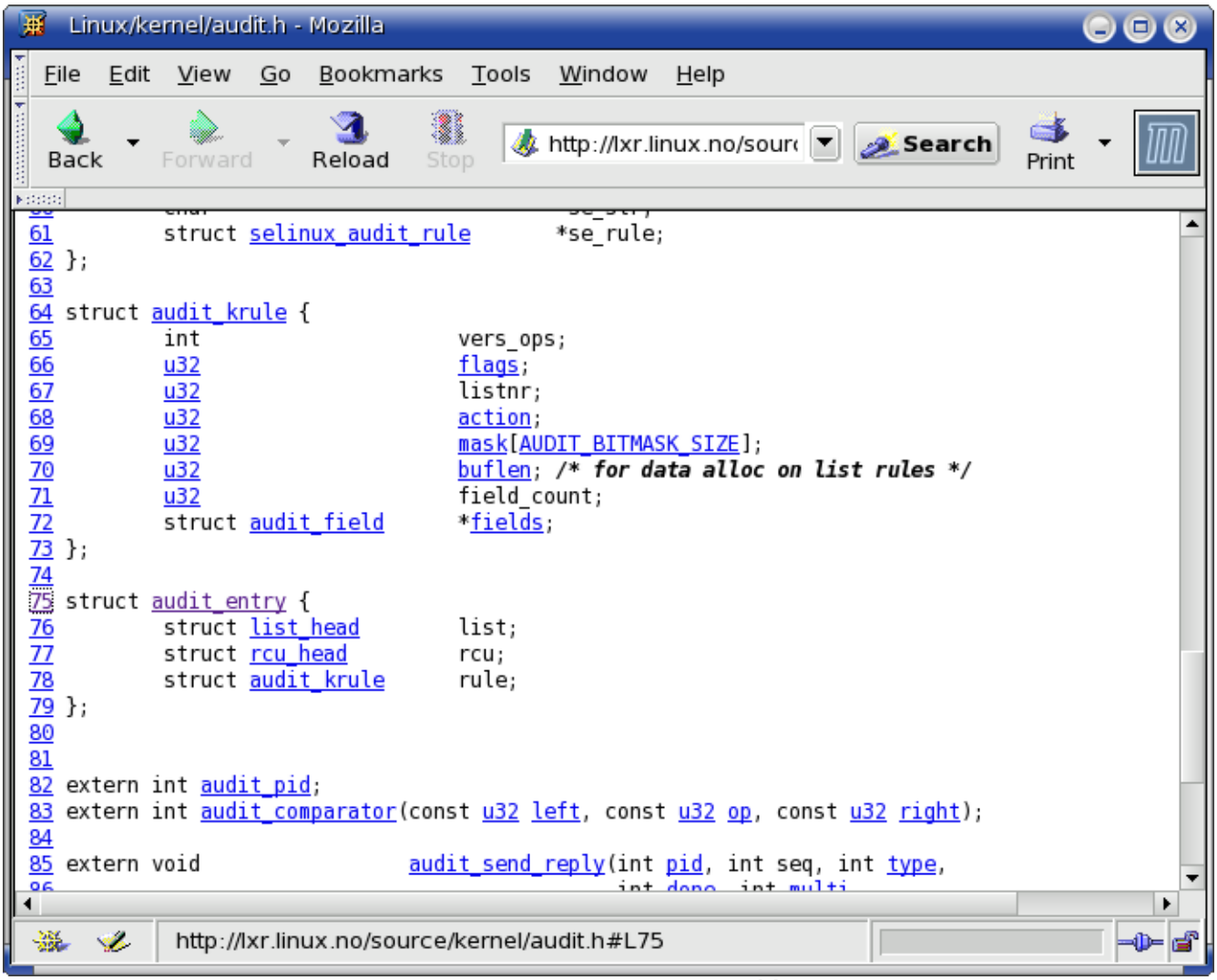


Figure 4: Definition of `audit_entry`.

LXR is a very powerful cross referencing tool. But it is also somewhat difficult to setup and get running. You'll need a working web server for the data generated. Also you'll need a good working knowledge of Perl in order to adapt the code for your system. (For your Perl hackers out there LXR contains some of the most complex, advanced regular expressions out there.)

But if you are dealing with a huge amounts of code this tool can be a lifesaver.

Chapter 6: Pre-processor hacks

C++ is actually two languages, a macro preprocessor language and a compiled language. So what you can't hack is the base language you can attempt to hack in the pre-processor.

Hack 67: Using the Pre-processor to Generate Name Lists

The problem: C++ does not have a way of getting the text form of a **enum**.

For example:

```
enum state {STATE_GOOD, STATE_BAD, STATE_UNKNOWN};

// Doesn't work right
std::cout << "Good is " << STATE_GOOD << std::endl;
```

The result is

```
Good is 0
```

What we want is

```
Good is STATE_GOOD
```

The Hack: Use the pre-processor to define a list of items in different formats

First we define a macro that contains the list of items we wish to use:

```
#define STATE_LIST \
    D(STATE_GOOD), \
    D(STATE_BAD),  \
    D(STATE_UNKNOWN)
```

The macro `D` (which we haven't defined yet) enclosed item. We will now use the D macro to define the list in a variety of ways:

```
#define D(x) x
enum state { STATE_LIST}
#undef D
```

Notice that we immediately undefine the `D` macro after it's use. This prevents this temporary macro from being accidentally reused later on in the code.

Now let's define a list of names.


```
#define D(x) #x
const char* const state_name = { STATE_LIST };
#undef D
```

We can now output the name of a state using a statement like:

```
std::cout << "The good state is " <<
     state_name[STATE_GOOD] << std::endl;
```

Limitations: This only works for `enum` lists where the things are in numerical order. If you assign values to items this will not work.

For example:

```
enum error_code {BAD=44, VERY_BAD=55, TERRIBLE = 999};
```

The next hack solves this problem.

Hack 68: Creating Word Lists Automatically

The Problem: You need to create a **enum** list with specific values.

For example: START is 5, PAUSE is 7 and ABORT is 25.

The Hack: A clever pre-processor hack can easily handle this.

Let's start by defining our list:

```
#define COMMAND_LIST      \
    C(COMMAND_START, 5),  \
    C(COMMAND_PAUSE, 7),  \
    C(COMMAND_ABORT, 25)
```

Now let's create a macro to define the **enum** declaration:

```
#define C(x,y) x = y
enum COMMANDS { COMMAND_LIST };
#undef C
```

The next step is to define the name to id number mapping:

```
#define C(x,y) {y, #x}

struct cmd_number_to_name {
    int cmd_number,
    char* cmd_name;
} cmd_number_to_name[] = {
    COMMAND_LIST
    , {-1, NULL}
```



```
};
#undef C
```

The use of nested macros like this is extremely useful when it comes dealing with data that must be expressed more than one way. Our hacks have concentrated on **enum** definitions, but this hack has other uses beyond these simple programming examples.

Hack 69: Preventing Double Inclusion of Header Files

The Problem: All header files should include any other headers they need. This can easily result in a single file being included multiple times. But defining the same thing multiple times can cause lots of problems.

The Hack: Add "double include" protection to your header files. For example:

```
// File define.h
#ifndef __DEFINE_H__
#define __DEFINE_H__
// ... rest of the header file
#endif /* __DEFINE_H__ */
```

Hack 70: Enclose Multiple Line Macros In do/while

The Problem: How do you create a macro that performs two statements.

For example we wish to create a cleanup macro:

```
#define CLEAN_RETURN \
    close(in_fd);close(out_fd); return;
```

But this code doesn't work if we put it in an **if** statement:

```
if (done)
    CLEAN_RETURN;
```

Expanding this we get:

```
if (done)
    close(in_fd);close(out_fd); return;
```

Let's add a little whitespace for clarity:

```
if (done)
    close(in_fd);
close(out_fd);
return;
```


This is not what we intended. One “solution” is to enclose the statements in {}.

```
#define CLEAN_RETURN \
    { close(in_fd);close(out_fd); return; }
```

Now our **if** statement expands to:

```
if (done)
    { close(in_fd);close(out_fd); return; };
```

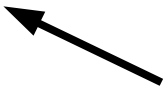
This works. Sort of. The problem is if we try an **if** / **else** statement:

```
if (done)
    CLEAN_RETURN;
else
    not_done_yet();
```

This gives us a syntax error when we try and compile it. Why?

Let's look at the expanded code:

```
if (done)
    { close(in_fd);close(out_fd); return; };
else
    not_done_yet();
```



There's an extra semicolon on the line. This didn't bother us when there was no **else**, but now that there is one, the compiler gets confused.

So how do we define a multi-statement macro that can be used like a statement?

The Hack: Use the **do** / **while** trick.

Define the macro so the statements are inside a **do** / **while** loop:

```
#define CLEAN_RETURN          \
    do {                      \
        close(in_fd);         \
        close(out_fd);        \
        return;               \
    } while (0)
```

Notice that there is no semicolon at after the `while (0)`.

Enclosing multiple statements in a **do** / **while** loop makes them a single statement. Thus they can be used anywhere a single statement can be used.

Note: This is probably the only place you legitimately want to use a **do** / **while**. In all other cases a **while** loop is probably simpler and easier to understand than a **do** / **while**.

Hack 71: Use #if 0 to Remove Code

The Problem: You need to make code disappear. Commenting it is one option, but that won't work if your compiler doesn't allow nested comments.

The Hack: Enclose the code in a **`#if 0`** / **`#endif`** block.

For example:

```
    do_it();
#if 0
    double_check_it();
#endif
    finish_up();
```

This hack takes the code out of your program. But the question remains, why not just delete the code. After all if you really want it to go away, delete it. Don't leave it in the program.

I've seen this hack used a number of times an each time I could see no reason for leaving the dead code in the program. But if you ever do have a reason for putting such code in a program file, this hack will keep it out of the program.

Hack 72: Use #ifndef QQQ to Identify Temporary Code

The Problem: Sometimes you just need to keep adding diagnostics to the code until you find the problem. Of course you don't want these temporary hacks showing in production code so it would be nice if you could remove them quickly.

The Hack: Put the code in a **`#ifndef QQQ`** / **`#endif`** block.

For example:

```
    read_account();
#ifndef QQQ
    save_backup("debug.1.save");
    std::cout << "Starting sort " << std::endl;
#endif /* QQQ */
    balance();
#ifndef QQQ
```



```
    save_backup("debug.2.save");
#endif /* QQQ */
```

The symbol **QQQ** was chosen because it's easy to type and no one defines it. (No one sane anyway.) Besides it's shorter than **MARKER_INDICATING_CODE_I_AM_TAKING_OUT_AS_SOON_AS_THIS_WORKS**.

The symbol makes the temporary code stand out like a sore thumb and it makes it easy to take the code out when you're done. Just search for **QQQ** and delete all the conditional code blocks associated with it.

Hack 73: Use #ifdef on the Function Not on the Function Call to Eliminate Excess #ifdefs

The Problem: You want to add some logging but only for the debug relase of the software. So throughout your code you have lots of stuff like this:

```
#ifdef DEBUG
    debug_msg("Entering function sixth_level_of_hell");
#endif /* DEBUG */
```

This is ugly and annoying. There must be a better way.

The Hack: If you define `debug_msg` right you can get rid of all those **#ifdef** statements and make our code clean. For example:

```
#ifdef DEBUG
extern void debug_msg(const char* const msg);
#else /* DEBUG */
static inline void debug_msg(const char* const) {}
#endif /* DEBUG */
```

Now in the body of the code all we have to just call the function:

```
    debug_msg("Entering function sixth_level_of_hell");
```

If `DEBUG` is defined, the real function is called. If it's not then the dummy `debug_msg` is called. Since it is an **inline** function, the optimizer will remove the code.

The nice thing about this is that you don't clutter up your code with useless **#ifdef** statements.

Hack 74: Create Code to Help Eliminate #ifdef Statements From Function Bodies

The Problem: You're writing a program for multiple platforms. As a result you have a lot of **`#ifdef`** statements sprinkled through out the code. So many in fact that they are make things look ugly and the code is hard to read.

For example:

```
void send_cmd(void)
{
    send_cmd_start();

#ifdef FE_TEXTURE
    send_texture();
#endif /* FE_TEXTURE */

#ifdef FE_COLOR
    send_background();
    if (foreground != TRANSPARENT)
        send_foreground();
#endif /* FE_COLOR */

#ifdef FE_SIZE
    if (size != 0)
         send_size();
#endif /* FE_SIZE */

#ifdef FE_REPLAY
     if (prev_cmd == '\0') {
         prev_cmd = cur_cmd;
         prev_param = cur_param;
     }
#endif /* FE_REPLAY */

    send_cmd_end();
}
```

This code is very difficult to read. The **`#ifdef`** directives breakup the logic of the code.

The Hack: Use **`#ifdef`** to define out entire procedures only.

For example, let's take a look at the code:


```
#ifdef FE_TEXTURE
    send_texture();
#endif /* FE_TEXTURE */
```

This can be much simpler. First we use **#ifdef** to control the definition of the function **send_texture**:

```
#ifdef FE_TEXTURE
static void send_texture() {
   // body of the function
}
#else
inline static void send_texture() { }
#endif /* FE_TEXTURE */
```

Now for the body of the code we just put in a call to **send_texture** without any **#ifdef**.

```
void send_cmd(void)
{
    send_cmd_start();
    send_texture();
```

We can do the same thing with **set_background** and **set_foreground**. The calls to these functions look like:

```
    send_background();
    if (foreground != TRANSPARENT)
        send_foreground();
```

But doesn't this cause extra code to be put in our procedure? What about the statement?

```
    if (foreground != TRANSPARENT)
```

Actually if the body of send_foreground is defined out, the optimizer will remove this statement.

Now let's look at the code.

```
#ifdef FE_REPLAY
     if (prev_cmd == '\0') {
         prev_cmd = cur_cmd;
         prev_param = cur_param;
     }
#endif /* FE_REPLAY */
```


To get rid of this **#ifdef** we first put the code in a procedure:

```
#ifdef FE_REPLAY
static inline void do_replay() {
     if (prev_cmd == '\0') {
         prev_cmd = cur_cmd;
         prev_param = cur_param;
     }
}
#else /* FE_REPLAY */
static inline void do_replay() {
    // Do nothing
}
#endif /* FE_REPLAY */
```

Now we can just put a function call in our code:

```
    do_replay();
```

The key to this hack is using **#ifdef** to change the definition of entire functions. Since functions are a simple, logical unit of code, we are defining things in or out at the unit level.

The old way changed the program at the statement level and was much more confusing. Take a look at how much simpler the body of our function is when we use this hack:

```
void send_cmd(void)
{
    send_cmd_start();
    send_texture();

    send_background();
    if (foreground != TRANSPARENT)
        send_foreground();

    if (size != 0)
         send_size();

    do_replay();
    send_cmd_end();
}
```


Chapter 7: Building Hacks

The size and complexity of programs is rising exponentially. As a result the build process is also growing in complexity and the time to produce builds is growing. The Linux kernel has over 13,000 files in it containing over 6.5 million lines of code. And to some people this is considered a small program.

Yet the basic build tools are still the *make* command and a compiler. But some hacks that let you get around the limits of this system and produce code quickly and sanely.

Hack 75: Don't Use any "Well Known" Speedups Without Verification

The Problem: There's no shortage of people who have a great idea about how to speed up the build. They have all sorts of reasons why their pet idea will let you complete a five day build in 3.8 seconds.

For example, the *gcc* compiler has the **`-pipe`** option. Normally the compiler will run the first pass of the compiler and write the output to a temporary file. Then pass 2 will read this file and write a second temporary. And so on for however many passes are needed.

With the **`-pipe`** option every pass is run at the same time the output of one connected to the input of the next through a pipe. The theory is that by doing this you eliminate the temporary file, the disk I/O for the temporary file and do everything in memory.

So the **`-pipe`** option must make things faster. And if we needed more convincing we can take a look at the kernel *Makefiles* and find that the option is used there and if the Linux kernel uses it, then it must be good.

The Hack: A good hacker tests the "obvious" because sometimes the "obvious" isn't obvious.

Let's take the case of the -pipe option. Compiling a kernel without the **`-pipe`** option takes 57:04. With the **`-pipe`** it takes 57:19. Wait a second, it took *longer* with **`-pipe`**. Let's do a few more tests:

No Pipe	*Pipe*
57:04	57:19
57:04	56:18
56:58	56:19
56:55	56:17
56:54	56:21

From this we can see that **`-pipe`** does not make things faster. If anything it makes them slower.

Now there could be a number of reasons for these results. First of all disk I/O is buffered and Linux uses lots of buffers. It could be that the temporary files exist entirely in memory and never make it to the disk. Also there could be different logic in the compiler passes for pipe / non-pipe input streams.

But ultimately as hackers all we're concerned with is results and the results do not support the theory.

There are lots of cases in programming where theory and reality do not match. Lots of people know where the slow parts of their code is. Hackers test by using a profiler.

A database manuals says that inserting records, then adding an index is faster than inserting indexed records. A hacker tests. Sometimes you find out that the manual is wrong.

After all one of the marks of a true hacker is someone who knows programming and computers frontwards and backwards. They not only know what the manual says with work, but also know if the manual is right or not.

The Congreve Clock

I am into horology[13] and one of my favorite types of clocks was designed by Sir William Congreve (1772 – 1828).

Sir William is best knows for his contribution to the *Star Spangled Banner*. If you remember the line “and the rockets red glare,” he invented the rockets.

13 It's a perfectly clean word – look it up.

During the time that he lived one of the great engineering challenges was making a accurate clock. Sir William was a prolific inventor and turned his genius to the problem of creating a highly accurate clock.

Ticking is one of the major sources of instability and inaccuracy in a clock. The movement of gears is a problem and the less they move the better. One way to solve this problem is by using a very long pendulum. In fact the longer the better.

Congreve came up with another solution. The result was the Congreve Clock or the Extreme Detached Escapement Clock as he called it. A ball is dropped on an inclined plane and allowed to roll down a track. When it reaches the end it trips a lever and the plane tilts the other way the the ball rolls down to the other end. This process can take 15 to 30 seconds depending on the clock.

Congreve calculated that his clock design was the equivalent or a conventional clock with a sixty foot pendulum. As a result his clock should be accurate to within a second a month.

Now William Congreve was a major inventor, engineer, and scientist. He had a great deal of logic and calculations to back up his claim. The design was perfect and the engineering impeccable. Extensive though and design went into creating the super accurate Congreve Clock.

Except it didn't work. Turns out that while a pendulums has a natural frequency a ball rolling down a path does not. Every speck of dust, every temperature change caused inaccuracy in the clock. And I won't even go into the other problems with his design. The goal was for a clock with an error rate of less than a second a month. But in fact a good Congreve Clock is only accurate to about twenty minutes a day.

Figure 5: A Congreve Clock

Hack 76: Use gmake -j to speed up compilation on dual processor machines

The Problem: Compiling big programs takes a long time. We need some way to make things faster. We have a dual processor machine. Is there any way to make the build faster other than to start two makes at the same time.

The Hack: Use the **`-j`** flag of *gmake* to make full use of a dual processor system.

Many modern computers today have more than one processor. Not only can you purchase high end motherboards which support 2 or 4 CPUs, but the both Intel and AMD produce dual core microprocessors which have the equivalent of two CPUs on a single chip.

The GNU *make* command (installed as *gmake* on most systems) has an option to help make optimal use of multiple processor systems. The **-j2** flag tell the *make* program to try and do two compiles at the same time. Ideally this should result in cutting the compile time in two.

There are some limitations to this system. First of all it's possible for compilation errors to appear out of order. For example, normally if out compile the files *alpah.cpp* and *beta.cpp* you might see:

```
alpha.cpp:3: Syntax error
alpha.cpp:7: Another syntax error
beta.cpp:3: Syntax error
beta.cpp:7: Screwed up again
```

With **-j2** you might see:

```
alpha.cpp:3: Syntax error
beta.cpp:3: Syntax error
alpha.cpp:7: Another syntax error
beta.cpp:7: Screwed up again
```

The other problem is that badly formed *Makefile*s may when you use **-j2** (or higher).

But is **-j2** really effective or is it just another **-pipe**? Tests show that on a dual processor system **-j2** actually does work. Here's the results:

No -j	***-j2***	***-j4***
56:19	29:47	29:38
56:18	29:43	29:45
56:19	29:48	29:43
56:17	29:42	29:42
56:21	29:44	29:45

Now just because it works on my system doesn't mean it will work on yours. But with a little experimentation you should be able to find the optimal *make* settings.

Extreme Parallel Processing

Hackers always want to know how the system behaves in extreme conditions. The **-j2** and **-j4** switches are rather mild. They will only start two or four jobs. What happens if we increase the number.

If no number is supplied and you just **`-j`** alone, *gmake* will start as many jobs as possible at the same time. I tried this once on a major piece of software. It was the only time I ever saw the load average go above 1,000[14]. I'm not sure how much higher the load average went because at that point the system monitoring tools started to hang. (They couldn't get any CPU cycles.)

The system was obviously paging like crazy and really beating up the disk. Obviously trashing[15] was going on and killing the system, so at this point I rebooted. (Couldn't kill the processes because the shell couldn't get enough CPU to make a difference.)

Hack 77: Avoid Recompiling by Using ccache

The Problem: Repeated compilation results in a lot of needless work.

We all know the drill. To create a clean build you need to execute the following commands:

```
make clean
make
make install
```

The reason you need a *make clean* is that sometimes some will chang e the *Makefile* and change something like flags or similar items. *These changes are not tracked by the* make *command.* As a result only sure way to get a good build is to recompile everything.

But most of your files have probably not changed since the last build. So you're basically reproducing the same file that you did the last time you compiled. This duplication of effort takes time and makes the build longer.

The Hack: Use the *ccache* program.

The *ccache* program caches compiled output and if you try to compile the same source again, gets the object file from the cache. The result is tremendously faster complication speeds at the expense of disk space. But given the cost of disk space today it's well worth the effort.

14 For the non-Linux types out there, the load average is a measure of how many jobs the system is trying to run at once. A load average of 0.1 to 3.0 is typical. Anything over 10 is considered extreme. A load average of 1,000+ is an indicator of both extreme load and extreme stupidity on the part of the user.

15 Thrashing is where the system spends more time switching resources around (disk, memory, etc.) than it does getting work done.

To use the *ccache* program simply insert the *ccache* command before each complication program in your *Makefile*s. For example:

```
hello.o: hello.cc
	ccache gcc -g -c hello.cc
```

When this program is executed *ccache* will perform the following operations:

1. Run the file *hello.cc through* the pre-processor.

2. Check the *md5sum* of the result against any stored checksum found in the cache directory.

3. If a match is found, the object is supplied from the cache.

4. Otherwise the compiler is run and the results stored in the cache for future use.

There are a huge number of details that have been glossed over, but that's the basic idea. The *ccache* program is a simple and very effective way of speeding up repeated builds.

The *ccache* program is available from http://ccache.samba.org.

Hack 78: Using ccache Without Changing All Your Makefiles

The Problem: You've got a legacy application with exiting *Makefile*s. You wish to use *ccache* but you don't want to edit every one of the 500 *Makefile*s to get the job done.

The Hack: Use the symbolic link feature of *ccache*.

The hackers who created *ccache* anticipated this program. If you create a symbolic link from the *ccache* program to the compiler. For example, if you use *gcc* you can enable *ccache* by executing the following commands:

```
$ ln -s /usr/local/bin/ccache /home/me/bin/gcc
$ PATH=/home/me/bin:$PATH ; export PATH
```

The first command creates a link between *ccache* and a *gcc* command in the local directory.

The second command makes sure that */home/me/bin/gcc* is the first *gcc* in the path. So when the *Makefile* executes *gcc* the command */home/me/bin/gcc* is run. This really *ccache* which is smart enough to know that it is being run through a symbolic link.

It then performs the normal operations just as if you and edited the *Makefile* and added a *ccache* command to each compilation rule.

Note: You probably want to add the *PATH* commands to your *.profile* or .bashrc so they are executed each time you start a new shell.

Hack 79: Distribute the Workload With distcc

The Problem: You have to compile a really big project and *ccache* is just not fast enough.

The Hack: If you have access to a lot of build machines you can use *distcc.*

The *distcc* program is available from http://distcc.samba.org/.

The tool is designed to work with the multiple job option of *gmake*. For example when you execute the command *gmake -j4* the *gmake* program will try and start four compilation jobs at once. With *distcc* in place each of these four jobs will be sent to a different machine.

The *distcc* program operates much like *ccache*. You activate it by putting *distcc* in front of your compilation commands. For example:

```
foo.o: foo.cpp
        distcc g++ -c -Wall foo.cpp
```

You'll also need to configure the system by supplying *distcc* with a list of machine it can use for building and installing *distcc* as a server on the build machines.

The *distcc* and *ccache* tools are designed to work together. So if you're faced with having to do repeated large builds, these tools can make your build much faster.

Chapter 8: Optimization Hacks

Hack 80: Don't Optimize Unless You Really Need to

The Problem: Premature Optimization

You have a program that works just fine. The program does not really use up a lot of CPU and gets the job done in a reasonable time. But if you optimize it you're sure you can shave a few seconds off the run time.

What do you do?

The Hack: Do nothing.

One of the hardest things for a programmer to do is to not program. But unless there is a compelling need for optimization, *don't do it!*

Simple working code always executes faster than optimized non-working code. Also simple working code is easier to maintain and enhance than clever, optimized code.

After all who cares if it takes 5 seconds instead of 3 to start your program. In most cases you program is probably not CPU bound anyway.

If you are doing large scale data manipulation such as video processing, creating virtual worlds, or large scale data compression then optimization makes sense. But most programs do not fall into these categories.

Leave working code alone. One of the surest ways of introducing a bug is to try and optimize something that doesn't need it.

Doing nothing is one of the best programming systems out there. It takes no time to write nothing, and nothing code is also the only code guaranteed to have zero bugs in it.

So the best advice on optimization is don't do it unless you really have to.

Hack 81: Use the Profiler to Locate Places to Optimize

The Problem: Where you think the slow spots in your code are located and where they really are located may be two different things.

Let's assume that you didn't take the advice in the previous hack and have decided that you must optimize. Now you must decide what part of your program you're going to try and make faster.

In most programs there are one or two functions which consume about 90% of the CPU. Many times a programmer will have a good feel for where his program is getting stuck. But more often than not it will turn out that his feeling does not agree with realist. As a result if he goes with his feelings he will wind up optimizing the wrong code and not speeding up his program.

The Hack: Use the Profiler

The profiler tells you which functions are taking up the most time.

In my experience a typical profiling session goes like this: I'm informed that the file save function is taking too long. I have an idea that the logic that converts the parameters to text for saving may be the program. After all the code was not designed to be efficient and it certainly fulfilled that design criteria.

So I start up my trusty profiler and try and locate the problem in the save logic. But something funny happens along the way. Turns out that save is taking only 1/10 of the time. The other 90% is in the sorting function.

Turns out that in order to get things ready for the save we sort the parameters. Just to make sure that they are sorted, every level of the code sorts them before calling the next lower level. In all there are twenty-five sorts done on the same data. The first sort puts things in order and the other twenty-four just waste time.

My choice at this point is sort less often or make sort more efficient. I decide to add a flag which tells me if the parameter list is sorted or not. The sort function checks that flag and does nothing if the list is in order. Problem solved.

Notice that the problem was not where I expected it to be. It was in some other code, which I didn't write by the way, which was terribly inefficient. Had I gone with my instinct and optimized the save code, I would not have made the program any faster.

The moral of this store is use the profiler. It tells you where the code really is slow rather than where you think it's slow.

Hack 82: Avoid the Formatted Output Functions

The Problem: C's formatted output functions are slow. C++'s formatted output functions are even slower.

Consider the following simple code fragment:

```
for (int i=0; i < 20; ++i) {
    printf("I is %d\n", i);
}
```

C and C++ are compiled languages and so are much faster than interpreted languages. But you've just added an interpreted language to your code. The interpreter is the `printf` call which must interpret the format string `"I is %d\n"` and send the result to the screen.

So what must `printf` do to print the result:

1. Read the string. Anything that's not a % goes to the output.
2. If a % is encountered, interpret the % specification.
   a. Check for any numbers after the % indicating the size of the string.
   b. Find the format character (in this case d)
3. Determine how many characters are needed to print the number.
4. If the number of characters needed to print the number is less than the number in the specification, then output leading spaces.
5. Output the number.

This is the simplified version. If you want to see the full code download a copy of the GCC standard library and look at the code yourself.

C++ formatted output is even worse. The library had a great deal of flexibility and lots of bells and whistles. All this general purpose baggage slows the library down greatly.

The Hack: Custom output routines


```
static void two_digits(const int i)
{
   if (i >= 10) {
      putc((i / 10) + '0');
   }
   putc((i % 10) + '0');
}

for (int i = 0; i < 20; ++i) {
    puts("I is ");
    two_digits(i);
    putc('\0');
}
```

It should be noted that the second, optimized, version is much longer and more complex than the first one. What we've done is replace slow general purpose code with special purpose code designed for this specific task.

It's faster, but it's more complex and more likely to contain bugs. Again only optimize if you must. (See Hack 80 above.)

Hack 83: Use ++x Instead of x++ Because It's Faster

The Problem: You need to increment a variable in the fastest possible way.

The Hack: Use the prefix increment (**++x**) instead of the postfix (**x++**) because it's faster.

Now a lot of you are probably saying to yourselves "You've got to be kidding". No matter which version I use to increment an integer the compiler is going to generate exactly code.

For example in the following code, the generated code is exactly the same for the two increments:

```
int i = 1;
++i;
i++;
```

For integers that is true. But it's not true for user defined classes.

Let's take a look at the steps needed to be performed for each operation.

The prefix (**++x**) increment must perform the following steps:

1. Increment the variable
2. Return the incremented result.

The postfix version (**x++**) performs the following:

1. Save a copy of the original value.
2. Increment the variable.
3. Return the unincremented copy you saved in step 1.

So for the postfix (**x++**) version you have to make a copy of the variable. This can be a costly operation.

Let's take a look at how a typical class might implement these two operations:

```
class fixed_point {
    // Usual stuff

    // Prefix (++x) operator
    fixed_point operator ++() {
        value += FIXED_NUMBER_ONE;
        return (*this);
    }

    // Postfix (x++) operator
    fixed_point operator ++(int) {
        // Extra work needed
        fixed_point result(this);
        value += FIXED_NUMBER_ONE;
        return (result);
    }
```

For this reason always use the prefix version of the increment and decrement operators to make your program more efficient.

Hack 84: Optimize I/O by Using the C I/O API Instead of the C++ One

It should be noted that this optimization is compiler dependent. However, every compiler I've used exhibits this behavior.

The Problem: The C++ I/O stream is more flexible and less error prone than it's C counterpart. But for most systems, it's also slower.

Consider a program designed to read a file line by line and write it to standard out.

The C++ version is:

```
#include <iostream>

int main()
{
    char line[5000];

    while (1) {
        if (! std::cin.getline(line, sizeof(line)))
            break;
        std::cout << line << std::endl;
    }
    return(0);
}
```

The C version is:

```
#include <stdio.h>

int main()
{
    char line[5000];

    while (1) {
        if (fgets(line, sizeof(line), stdin) == NULL)
            break;
        fputs(line, stdout);
    }
    return (0);
}
```

It takes 98 seconds for the C++ program to copy a bunch of files to */dev/null* while it takes the C program only 11 seconds to do the same thing.[16]

The Hack: Use C I/O in a C++ program when I/O speed is an issue

This hack does speed up I/O bound programs, but it should be used with caution. First you do loose the advantages of the C++ I/O system. This includes the increased safety that comes from this system.

16 On a Linux system using GCC Version 3.4,3, But you're a hacker – run your own tests.

Also the timing tests presented here come from one compiler on one computer system. You should perform similar tests using your programming environment. Who knows your compiler may be better.

Hack 85: Use a Local Cache to Avoid Recomputing the Same Result

The Problem: You have a function that is called repeatedly to compute the same values. This function is taking a long time to do its job. How can you make things faster?

Let's take a look at an example. Below is a **`rectangle`** class. The function **`compute_serial`** returns a unique serial string which is used for sorting the shapes classes.

```
class rectangle {
    private:
        int color;
        int width;
        int height;
    rectangle(int i_color, int i_width, int i_height):
       color(i_color), width(i_width), height(i_height)
    {}

    void set_color(int the_color) {
         color = the_color;
    }

    char serial_string[100]; // Unique ID string

    char* get_serial() {
        snprintf(serial_string, sizeof(serial_string),
             "rect-%d x %d / %d", width, height, color);
        return (serial_string);
    }
```

Examination of this class shows that it uses the **`snprintf`** to compute the serial string. This is an extremely costly function to use. We need optimize out this function if at all possible.

The Hack: Use a local cache.

A small local cache can be very useful when it comes to avoid the problem of computing the same thing over and over again.

For our **rectangle** class this is accomplished with some simple changes to **get_serial** function:

```
char* get_serial() {
    if (dirty) {
        snprintf(serial_string, sizeof(serial_string),
             “rect-%d x %d / %d”,
             width, height, color);
        dirty = false;
    }
    return (serial_string);
}
```

Now all we have to do is make sure that **dirty** is set any time the data used to compute the serial_string is changed. This includes the constructor:

```
rectangle(int i_color, int i_width, int i_height):
   color(i_color), width(i_width), height(i_height),
   dirty(true)
{}
```

Any function which changes any of the key members must be changed as well:

```
void set_color(int the_color) {
     color = the_color;
     dirty = true;
}
```

This hack trades complexity for speed. By making the class more complex, we can make it faster.

So this hack should be used with care. You should only use it *after* you've run your program through a profiler and made sure that this class is one of the bottlenecks. Far too often people who think they are hackers optimize a program where they think there's a speed problem only to find out that they are not real hackers and don't have a speed problem where they think they do. Profilers identify real speed problems, that's why they are used frequently by real hackers.

There is a price to be paid for using this hack. A real hackers knows when the price is worth it.

Hack 86: Use a Custom new/delete to Speed Dynamic Storage Allocation

The Problem: You have a small class that is frequently allocated and deallocated from the heap. The dynamic memory allocation calls are really slowing down your program. How can you speed things up?

The Hack: Make a custom **new** / **delete** for your class.

I should point that this hack is tricky and not easy to implement. You should only use it if you're sure you know what you're doing and you're sure that **new** and **delete** are a problem for this class. Implementing this wrong can corrupt memory or cause lots of other problems, so make sure you've read hacks Hack 80 and Hack 81 before you start.

To create your own **new** for a class all you have to do is to declare it as a local operator function. Here's an example:

```
// A very small class (good for local new / delete)
class symbol {
    private:
        char symbol_name[8];
    // .. usual member functions
    void* operator new (unsigned int size) {
        if (size == sizeof(symbol)) {
            void* ptr = local_allocate(size);
            if (ptr != NULL) {
                return (ptr);
            }
        }
        return (::new char[size]);
    }
```

Let's go through this step by step. First we have the function declaration:

```
    void* operator new (unsigned int size) {
```

This defines a operator **new** that overrides the one built-in to C++. It takes one parameter, the size of the item to be allocated.

One mistake people make is that they assume that the when you create a object who's type is **`class symbol`** that the size of the object is going to be **`sizeof(class symbol)`**. It's not. If if **`symbol`** is the base class in a derived class definition (say **`class extended_symbol`**) then when a **`extended_symbol`** is allocated, the **`operator new`** of **`symbol`** is called with the size of **`extended_symbol`**.

If you do not understand exactly what I just said, the don't do this optimization until you do. I suggest you perform some experiments with custom **new** / **delete** operator using both base and derived classes. Do not try this hack in production code unless you fully understand what is going on!

We are going to assume that program allocates mostly variables of type symbol and that's what we want to optimize. But just in case someone decided to extend our class, we check the size.

```
        if (size == sizeof(symbol)) {
```

If the size is not right, we'll fall through to the bottom of the program and use the system new to allocate the memory.

```
        return (::new char[size]);
```

If the correct size is given to use, that means we are allocating a new **`symbol`** so we can use the optimized memory allocation:

```
        if (size == sizeof(symbol)) {
            void* ptr = local_allocate(size);
```

We'll leave the implementation of **`local_allocate`** to you. It should be simple and fast. (If it's big a slow, what's the point of using it?) For example, the allocator could use a fixed memory array to hold the data. Since you have a fixed size item and know something about the expected allocate / free usage you should be able to make a very efficient memory manager.

Whenever we do a custom **new** we need to do a custom **delete** as well:

```
    static void delete(void* const ptr) {
        if (local_delete(ptr))
            return;
        ::delete ptr
    }
```

One final note, read the next anti-hack before proceeding.

Anti-Hack 87: Creating a Customized new / delete Unnecessarily

The Problem: The programmer *thinks* his program is slow and *feels* that a local **new / delete** will speed things up.

The Anti-Hack: Optimize your program by putting in your own **new / delete**.

Most of the time if you add in an optimization without running the program through a profiler and carefully analyzing it, you will optimize in the wrong place. The **new / delete** hack is tricky and dangerous. You don't want to add it unless you *know* that it will speed up your program. That means you have to do your homework.

Far too often a person who thinks he's a hacker, but isn't will perform the local **new / delete**. A predictable series of results generally follows.

First the code breaks. That's because getting local **new / delete** operators to function right is tricky. (See the previous hack for details.) Because the programmer has implemented **new / delete** wrong his program fails in mysterious ways. Tracking these down takes time. Eventually he will give up or finally get a working local **new / delete**. Of course the project will fall behind schedule during this process.

After he finally gets the new memory management system debugged, he will discover that because he didn't do his homework and run a profiler, he just sped up class that not part of the time critical program flow. Thus the result is that the program is not any faster than when he started.

So the only thing he has to show for his effort is a skipped schedule. But what did he gain? He learned a important lesson in premature optimization. A lesson you don't have to learn the hard way because you read this anti-hack.

Anti-Hack 88: Using shift to multiple or divide by powers of 2

The Problem: Some people use left shift to multiple by 2, 4, and other powers of 2. They use right shift to divide by the same numbers.

For example the following two lines do the same thing:

```
i = j << 3;
i = j * 8;
```

The Anti-Hack: Use shift to speed up calculations.

Real hackers multiply by 8 when they want to multiply by 8.

Back in the days when compilers were stupid and machines slow, if you multiplied by 8 the compiler generated multiply instructions. This could be quite slow as some machines didn't have a multiply instruction, so in order to perform this operation, a subroutine call was used.

People quickly discovered that you could use shift instead of multiply (or divide) when using a power of 2 (2,4,8,16....). The resulting code was much faster.

But machines have improved and so has compiler technology. If you need to do a multiply, do a multiply. The compiler will generate the fastest code possible to accomplish this operation.

All you do when using a shift instead of a multiply or divide is making your code more obscure. We've got way too much obscure code now, be clear and simple. Use multiple to multiply and divide to divide.

Hack 89: Use static inline Instead of inline To Save Space

The Problem: You need to save a few bytes.

We'll ignore the question as to way these few bytes would make a difference given the current memory memory sizes and prices. Let's just say you need to make memory usage as small as possible.

The Hack: Always declare **inline** functions **static**.

The **inline** directive tells the compiler to generate the code for the function in-line. That works fine for functions calls inside the file where the function is declared.

But what happens in the following case:

```
File sub.cpp
inline int twice(int i)
{
   return (i*2);
}

File body.cpp
extern int twice(int i);

int j = twice(5);
```

In this case it's perfectly legal for the file *body.cpp* to contain a call to **`twice`**. But *body.cpp* does not know that the twice is an **inline** function. It's going to call it just like a normal **extern** function.

So the compiler is forced to not only generate all the in-line instances of **twice**, but also generate standard function code just in case someone from outside tries to call it.

Declaring the function **static** tells the compiler that no one from the outside is going to call this function and let's it safely eliminate the standard function code. Thus saving a few bytes.

Hack 90: Use double Instead of Float Faster Operations When You Don't Have A Floating Point Processor

The Problem: You are writing code for an embedded system and you have to use real numbers. Is there a simple way of speeding things up?

The Hack: Use **double** instead of **float**. It's faster.

At first glance this advice seems silly. Everyone knows that **double** contains more bits than **float** therefore it is "obvious" that using **double** is going to take longer than **float**. Unfortunately the words "obvious" and "programing" frequently incompatible.

The C and C++ standard states that all real arithmetic is to be done in **double**. Let's take a look at what goes on under the hood of the following code:

```
float a,b,c;
a = 1.0; b = 2.0;

c = a + b;
```

The steps needed by the addition are:

1. Covert **a** to **double**.
2. Covert **b** to **double**.
3. Add the **double** version of **a** to the **double** version of **b**.
4. Convert the result to **float**.
5. Store the result in **c**.

Conversion from **float** to **double** and **double** to **float** are not cheap and all the conversions take time.

Now let's look at what goes on when we change the variables to **double**.

1. Add **a** to **b**.

2. Store result in **c**.

Three **float** / **double** conversions have been removed. Since a floating point conversion operation is expensive, the result is that the code using **double** is much faster than **float** version.

Be warned, this hack will not work on all machines. In particular it will not work on PCs and other machine which have a floating point processor. (That's why we said we were doing an embedded program when we described the problem.)

Hardware floating point units almost always convert any numbers to an internal format before doing the arithmetic no matter format is used. (Some compilers use the **long double** type to represent this format.)

Smaller, embedded processors don't have the luxury of floating point units so they use a software library for their floating point. If this is the case for the system you're working on, this hack will probably work.

You should always test this hack before trying it. Floating point operations have a life of their own and can do strange things to unsuspecting code.

Hack 91: Tell the Compiler to Break the Standard and Force it To Treat float as float When Doing Arithmetic

The Problem: You have to use floating point on an embedded system and can afford the extra storage needed to store all the values a **double**. How can you speed things up?

The Hack: Tell the compiler to break the standard.

Read your compiler documentation. Most compilers have an option that lets you generate single precision arithmetic even though the standard calls for double precision.

For the gnu *gcc* compiler this option is **`-fallow-single-precision`**. When this option is enabled then adding two single precision numbers is done by adding the two numbers. Not converting, adding, and converting back.

Turning this on makes your program execute much faster with very little loss of precision.

Again, as we've discussed in the last hack, this only works when you are *not* using a machine with a floating point processor. When hardware is used, the hardware does it's own thing concerning conversions, so the compiler option has no effect.

Hack 92: Fixed point arithmetic

The Problem: You are coding a graphics filter that process each of an image through a filter. Each pixel is defined a triplet of real numbers (R,G,B) each of which is between 0 and 255.

The filtering algorithms need to work on real numbers in order to do their job. But examining the program you discover that they only need two digits of accuracy. Any additional digits is just overkill.

The Hack: Fixed point arithmetic.

Fixed point arithmetic lets you do calculations with real numbers just like floating point. The difference is that you can't move the decimal point. It's fixed at two digits.

The advantage is that you can implement fixed point arithmetic using integers. Integer arithmetic is much much faster than floating point.

In this example we only need the values 0.00 to 255.00 so we can store the data in a **short int**.

```
    private:
        // The value of the number
        short int value;
```

To assign our fixed point number a floating point value, all we have to do is multiply by a conversion factor. This converts the real value to an integer suitable for the integer we are using to represent the number.

```
fixed.value = real_number * 100;
/* 100 because 2 decimal places */
```

To add and subtract a fixed point number all we have to do is add or subtract the internal value. Multiply is a little more tricky. We need to apply a conversion factor to move the implied decimal point back to the right place:

```
f_result = f1 * f2 / 100;
```

For division the conversion factor is multiplied.

```
f_result = (f1 / f2) * 100
```


The math is actually fairly simple. Unfortunately because of C++ rich operator set, you have to define a lot of operators. The code at the end of this hack contains a *limited* implementation of a simple fixed point class.

One final note, in this example, we use a fixed decimal point two places to the left. The system would be a lot faster if we used a fixed *binary* point 7 places to the left. In other words, a conversion factor of 128 is much faster than 100. It gives us the same precision at slightly better speed.

```
#include <ostream>

class fixed_point {
    private:
        // Factor that determines where
        // the fixed point is
        static const unsigned int FIXED_FACTOR = 100;

        // Define the data type we are using for
        // the implementation
        typedef short int fixed_implementation;

    private:
        // The value of the number
        fixed_implementation value;

    public:
        // Default value is 0 when
        //  constructed with no default
        fixed_point() : value(0)
            {}
        // Copy constructor
        fixed_point(const fixed_point& other) :
             value(other.value)
            {}
        // Let the user supply us a double as well
        fixed_point(double d_value) :
            value(
                static_cast<fixed_implementation>(
                   d_value /
                   static_cast<double>(FIXED_FACTOR))
            )
            {}

        // Let the user initialized with an integer
        fixed_point(int i_value) :
            value(i_value * FIXED_FACTOR )
```



```
            {}

        fixed_point(long int i_value) :
            value(i_value * FIXED_FACTOR )
            {}

        // All the usual assignment operators
        fixed_point& operator =
                      (const fixed_point& other) {
                value= other.value;
                return *this;
            }

        fixed_point& operator = (float other) {
            value = fixed_point(other).value;
            return *this;
        }

        fixed_point& operator = (double other) {
            value= fixed_point(other).value;
            return *this;
        }

        fixed_point& operator = (int other) {
            value= fixed_point(other).value;
            return *this;
        }

        fixed_point& operator = (long other) {
            value = fixed_point(other).value;
            return *this;
        }

        // Conversion operators
        operator double() const {
            return static_cast<double>(value) /
                   static_cast<double>(FIXED_FACTOR);
        }
        operator short int() const {
            return value / FIXED_FACTOR;
        }
        operator int() const {
            return value / FIXED_FACTOR;
        }
        operator long() const {
```



```
            return value / FIXED_FACTOR;
        }

        // Unary operators
        fixed_point operator +() const {
            return *this;
        }
        fixed_point operator -() const {
            fixed_point result(*this);
            result.value = -result.value;
            return result;
        }

        // Binary operators
        fixed_point operator + (
                    const fixed_point& other) const {
            fixed_point result(*this);
            result.value += value;
            return (*this);
        }
        fixed_point operator - (
                    const fixed_point& other) const {
            fixed_point result(*this);
            result.value -= value;
            return (*this);
        }
        fixed_point operator * (
                    const fixed_point& other) const {
            fixed_point result(*this);
            result.value *= value;
            result.value /= FIXED_FACTOR;
            return (*this);
        }

        fixed_point operator / (
                    const fixed_point& other) const {
            fixed_point result(*this);
            result.value *= FIXED_FACTOR;
            result.value /= value;
            return (*this);
        }

        // Comparison operators
        bool operator == (
                      const fixed_point& other) const {
```



```
            return value == other.value;
        }
        bool operator != (
                    const fixed_point& other) const {
            return value != other.value;
        }
        bool operator <= (
                    const fixed_point& other) const {
            return value <= other.value;
        }
        bool operator >= (
                    const fixed_point& other) const {
            return value >= other.value;
        }
        bool operator  < (
                     const fixed_point& other) const {
            return value  < other.value;
        }
        bool operator  > (
                     const fixed_point& other) const {
            return value  > other.value;
        }

        friend std::ostream& operator << (
                    std::ostream& out,
                    const fixed_point& number);

};

// Quick and dirty output operator.
std::ostream& operator << (
         std::ostream& out,
         const fixed_point& number) {
    out << static_cast<const double>(number);
    return (out);
}
```

Hack 93: Verify Optimized Code Against the Unoptimized Version

The Problem: We've decided that Hack 92 works for us. But how can we tell if we've implemented the class correctly?

The Hack: When replacing one system with another, test twice and compare results.

In this Hack 92 we started with pixels represented by **float** and replaced it with a **`fixed_point`** implementation. Both versions should produce the same result.

But as hackers we know that there's a vast difference between “should” and “is”. That's where testing comes in.

A good test to see if the optimization was correct is to run the unoptimized program and save the results. Then run the optimized version and check to see that we get the same results.

In real life Hack 92 was used to speed up a sophisticated dithering algorithm for a color inkjet printer. When a binary comparison was done on the output the result were *different*. So in spite of what our high powered numerical analysis expert said, two digits were not enough to generate identical results.

However when the test images were printed, the optimized results looked just as good to the Print Committee as the unoptimized versions. So the computer could tell the difference between the two algorithms but the Print Committee could not. In the inkjet business, the Print Committee rules so we kept the newer, faster algorithm.

There's a moral to this story and as soon an I figure out what it is I'll put it in this book.

Case Study: Optimizing bits_to_bytes

Let's take a look how a hacker would optimize a simple function. The job of this function is to figure out how many bytes it takes to store a given number of bits. Here's the function as it first appeared:

```
short int bits_to_bytes(short int bits)
{
    short int bytes = bits / 8;

    if ((bytes % 8) != 0) {
        bits++;
    }
    return (bytes);
}
```

The first thing we notice about this function is that the code is pretty bad. As hackers we are frequently faced with horrible code.

I once optimized a program that was taking 20 hours per run down to the point where it was taking about 8 seconds per run. Now I'm a good hacker, but I'm not that good. The original program was very badly written. In defense of the original programmer, it was the first program he had ever written and he did a remarkably job of implementing a sophisticated cryptographic algorithm despite not knowing many basic features of the C language.

The **bits_to_bytes** function was targeted for a cell phone (ARM processor). Running the code through the compiler and taking a look at the assembly code we find some interesting things going on. For example the implementation of the line:

```
    short int bytes = bits / 8;
```

The generated code looks like:

```
    movw r1, bits        ; r1 (bytes) = bits
    asr r1, 3            ; r1 = r2 >> 3 (aka r1 = r1/8)
    lsl r1, 16           ; r1 = r1 << 16 (?)
    asr r1, 16           ; r1 = r1 >> 16 (?)
```

What's going on with the two funny instructions that shift the result left by 16 then right by 16? At first glance this is a rather useless piece of code.

The processor uses 32 bit arithmetic. On this machine a **short int** is 16 bits. When the system does the divide by 8 a 32 bit result is generated. So the compiler generates two instruction designed to convert a 32 bit value into a 16 bit one.

This occurs after the divide and after the increment. This gives us a total of 4 useless instructions.

Changing the function to use **int** instead of **short int** eliminates these instructions and makes our code faster.

The next step is to see if we can write a better algorithm and eliminate that conditional:

```
int bits_to_bytes(int bits)
{
    return (bits + 7) / 8;
}
```

This completely eliminates all the conditional logic an saves us a few more instructions. Now the actual body of the function is just a hand full of instructions. Can we cut it down even more?

How about cutting it down to 0 instructions? It's possible. All we have to is to add the **inline** keyword:

```
inline int bits_to_bytes(int bits)
{
    return (bits + 7) / 8;
}
```

Now when the optimizer sees a line like:

```
    store_size = bits_to_bytes(41);
```

it will optimize it down to;

```
    store_size = 6;
```

The function is not even called. All the computations are being done at compile time.

But we're not done optimizing. There's a difference between a function declared **inline** and one declared **static inline**. When a function is declared **inline** the compiler will inline all the function calls it sees, *then generate a regular non-inline function body in case someone from the outside wants to call this function.*

To counter this we declare our function **static inline** and stick it in a header file. Thus saving us a couple of dozen bytes in this example – total – in a 3.5MB program.

Now as hackers there's one more thing we need to consider. What happens when things go wrong. After all we never trust the caller to do things right and the we can be called with a negative number. We need to answer the question "How much storage does **-87** bits take up?"

The easiest thing to do is to assume that this will never happen and just ignore it. But if we do this we need to document the fact in the program.

```
/*
 * bits_to_bytes – Given a number of bits, return the
 *     number of bytes needed to store them.
 *
 * WARNING: This function does no error checking so
 * if you give it a very wrong value you get a very wrong
 * result.
 */
```

Assuming something bad will never happen is not really a good idea. As hackers we know that lots of things that can "never happen" actually do. Checking for the "impossible" usually a good idea.

For example, we could insert an **assert** statement:

```
static inline int bits_to_bytes(int bits)
{
    assert(bits >= 0);
    return (bits + 7) / 8;
}
```

The problem with **assert** statements is that they cause the program to abort. In real life this code lived in a mobile phone and a failed **assert** would cause the phone to reset. This was not good because when this happened the phone reset it would play the “welcome sound”. End users were wondering why their phone would restart for “no reason at all”.

The phone maker's “solution” to this problem was simple. They changed the code so that a a failed **assert** restarted the phone silently. The code was still very buggy, but the bugs became less visible (audible?) to the end users.

Also you should remember that assertions can be compiled out, so this code provides no protection at all.

Since this is C++ throwing an exception is one way of handling the error:

```
static inline int bits_to_bytes(int bits)
{
    if (bits < 0) throw(memory_error(“bits_to_bytes”);
    return (bits + 7) / 8;
}
```

These error checking options are expensive. One inexpensive thing we can do is to make sure that error can never happen. How do we do that? All we have to is make the argument (and the return value) **unsigned.**

```
static inline unsigned int
     bits_to_bytes(unsigned int bits)
{
    return (bits + 7) / 8;
}
```

Finally there's one more way of dealing with this error. Just change the function to silently ignore it and return a default value:

```
static inline int bits_to_bytes(int bits)
{
    if (bits < 0) return (0);
    return (bits + 7) / 8;
}
```


This is usually not a good idea since such code tends to hide errors in other pieces of code. In general you don't want to silently fix things. Maybe output a log message:

```
static inline int bits_to_bytes(int bits)
{
    if (bits < 0) {

        log_error(
            “Illegal parameter in bits_to_bytes(%d)”,
            bits);

        log_error(“Standard fixup taken”);
        return (0);
    }
    return (bits + 7) / 8;
}
```

In examining bits_to_bytes we can see that it is actually a very short function. But it does illustrate some of the things good hackers consider when working with code. These include:

- Dealing with lousy code
- Knowing how the compiler generates code and designing your code to make optimal use of this information.
- Making maximum use of the language features such as **inline** and **static inline**.
- Being paranoid[17]. Deciding what to do with bad data.

There's a lot to be learned from this little program. As hackers we're always learning. We study, research, experiment, and play all to gain a better understanding of the programming process.

17 Just because you're paranoid doesn't mean they aren't out to get you.

Chapter 9: g++ Hacks

The GNU *gcc* package is one of the most C and C++ compilers out there. Two things lead to its popularity. First it's a high quality compiler. The second is that it's free.

The GNU compiler has extended the C and C++ languages in some useful ways (and some useless ones too). If you are willing to sacrifice portability the non-standard language can be very useful.

Hack 94: Designated Structure Initializers

The Problem: You have a structure with 2,000 elements in it[18]. To initialize the structure you need to specify 2,000 items *in exact order*. It's easy to make a mistake.

For example:

```
struct very_large {
    int size;
    int color;
    int font;
    // 1,997 more items
};
// Now create a instance of item
very_large something_big = {
    SIZE_BIG,
    FONT_TIMES,
    COLOR_BLACK,
    // 1,997 more items;
};
```

If you spotted the mistake in the previous example, congratulations. If not, then you know why this hack is so needed.

The Hack: Use the *gcc* designated structure initializer syntax.

18 For the humor impaired, I'm exaggerating. A real hacker would split up something this big into multiple smaller and more manageable structures.

The *gcc* compiler has a non-standard extension that lets you name the fields in an initialization statements. For example:

```
very_large something_big = {
    .size = SIZE_BIG,
    .font = FONT_TIMES,
    .color = COLOR_BLACK,
    // 1,997 more items;
};
```

By naming things we decrease the risk of getting something in the wrong order. The problem is that we have created a non-standard program. But unless your dealing with a very unusual platform, there is a *gcc* compiler for your system.

But there is one problem with this system. It is possible to omit a field. If you do it gets a default value (0). But what happens if you accidentally omit a value. It gets assigned a default and without warning unless you tell *gcc* to warn you. The option `-Wmissing-field-initializers` tells *gcc* that you want to specify every field when you initialize. If you don't a warning will be issued.

Hack 95: Checking printf style Arguments Lists

The Problem: You've written a function called `log_error` which uses `printf` style arguments. How can you tell *gcc* to check the arguments just like it does for `printf`?

The Hack: Use the `__attribute__` keyword to tell *gcc* what to check. For example:

```
void log_error(const int level,
    const char* const format,
    ...) __attribute__((format(printf, 2, 3)))
```

(Yes the double parenthesis are needed.)

The `__attribute__(format...)` syntax tells *gcc* to treat the arguments to log_error just like `printf`. In this case the function is like *printf*, the format string is the second (2) argument to the function and the parameters start with position 3.

If you have warnings turned on, *gcc* will now check the parameter lists of any call to `log_error` to see if the parameters match the format.

Note: You may wish to put the following lines in your code if you wish to compile using both *gcc* and non-*gcc* compilers.


```
#ifndef GCC
#define __attribute__(x)    /* Nothing */
#endif /* GCC */
```

Hack 96: Packing structures

See Hack 116.

Hack 97: Creating Functions Who's Return Shouldn't Be Ignored

The Problem: Some functions return a value that never should be ignored. For example, there is no reason that anyone would want to call **`malloc`** and ignore the results. Is there any way to force people to use the return value of a function?

The Hack: Hackers don't try to force people to do things. Instead they use persuasion. Peer pressure can be a great motivator when it comes to getting people to do things right.

In this case we can make things easier by making use of the *g++* compiler's **`__attribute__`** feature. Setting the **`warn_unused_result`** attribute causes the compiler to issue a warning when the return value of the function is ignored.

Here's a short example:

```
#include <malloc.h>
#include <cassert>

void* my_malloc(size_t size)
    __attribute__((warn_unused_result));

void* my_malloc(size_t size) {
    void* ptr = malloc(size);
    assert(ptr != NULL);
    return (ptr);
}
```

Now let's use this function. Of course we'll use it wrong and just throw the result away creating a memory leak. Normally this would be a problem:

```
int main()
{
    // Memory leak
    my_malloc(55);
```



```
    return (0);
}
```

But since we used **__attribute__((warn_unused_result))** the compiler will complain about this code:

```
result.cpp: In function `int main()':
result.cpp:16: warning: ignoring return value of `void*
my_malloc(size_t)', declared with attribute
warn_unused_result
```

Hack 98: Creating Functions Which Never Return

The Problem: We've created a function called **die** which prints an error message and aborts the program. Yet when we use it the compiler keeps giving us warnings.

For example:

```
#include <iostream>

void die(const char* const msg)
{
    std::cerr << "DIE: " << msg << std::endl;
    exit(8);
}

int compute_area(int width, int height) {
    int area = width * height;

    if (area >= 0) return (area);

    die("Impossible area");
}
```

The problem is that the compiler keeps complaining about this code:

```
die.cpp: In function `int compute_area(int, int)':
die.cpp:15: warning: control reaches end of non-void
function
```

The Hack: Use the *g++* **noreturn** attribute.

For example:

```
void die(const char* const msg)
   __attribute__((noreturn));
```



```
void die(const char* const msg) {
// ... function body
```

This tells *g++* that the function never returns. The functions **abort** and **exit** have this option set. Since we've told *g++* that **die** will never return, the compiler will no longer fuss when compiling the previous example.

If we turn on the unreachable code warning (**-Wunreachable-code**) the compiler will fuss if we try and put code after the **die** call as it should. For example:

```
#include <iostream>

void die(const char* const msg)
    __attribute__((noreturn));

void die(const char* const msg)
{
    std::cerr << "DIE: " << msg << std::endl;
    exit(8);
}

int compute_area(int width, int height) {
    int area = width * height;

    if (area >= 0) return (area);

    die("Impossible area");
    return (0);    // Return a default value
}
```

When compiled we get the error:

```
g++  -Wunreachable-code -Wall -c die.cpp
die.cpp: In function `int compute_area(int, int)':
die.cpp:18: warning: will never be executed
```

This should give the programmer who added the default return value an idea that maybe his code is not going to do what he expected. It's always good when the compiler warns you about strange code and as hackers we want to use the compilers code checking to maximum advantage. The **__attribute__** hacks let us do just that.

Hack 99: Using the GCC Heap Memory Checking Functions to Locate Errors

The Problem: Memory corruption. I'm not going to go into details. If you're an experienced programmer, you've gone through your share of horrible memory problems.

The Solution: Use the GNU C library (*glibc*) memory checking functions to help locate problems.

To turn on the memory checking functions you need to put the following code in your program:

```
#include <mcheck.h>

// ....

int main()
{
    mcheck_pedantic(NULL);
```

The **memcheck_pedantic** call must occur before any allocations are done in order to turn on memory checking. The argument is a pointer to a function which is called when an error is detected. If this parameter is **NULL** then the a default function is used. This function prints an error and aborts the program.

The **memcheck_pedantic** function causes the memory allocation functions to put sentential at the start and end of each allocated block. Ever time you call **malloc**, **free**, or other memory allocation function all these sentential will be checked.

When a problem is detected the program will abort.

For example:

```
#include <malloc.h>
#include <mcheck.h>

int main()
{
    mcheck_pedantic(NULL);
    char* ptr = (char*)malloc(10);

    ptr[10] = 'X';
    free(ptr);
    return(0);
```



```
}
```

When run this program prints:

```
memory clobbered past end of allocated block
```

The **mcheck_pedantic** performs a consistency check every time memory is allocated or freed. Every memory block is checked. As a result your program will run slower than normal.

There is a faster memory check function, **mcheck**. It turns on pointer checking only for the pointer being allocated, reallocated, or freed. You can force the system to check all the memory pointers by calling **mcheck_check_all**.

A single pointer can be checked by the **mprobe** function.

Almost all of these functions are documented in the glibc documentation. (Use the command *info glibc* on most Linux distributions.) The one exception is **mcheck_pedantic**. It's not mentioned in the documentation at all.

However it is mentioned in the header file (*mcheck.h*). So another lesson we can learn from this hack is to always take a look at the header files to see if we can find hidden treasure.

Hack 100: Tracing Memory Usage

The Problem: Memory leaks. (And it's not feasible to use *valgrind* – See Hack 41.)

The Solution: Use the **mtrace** function to trace all memory allocations and deallocations.

In order to start memory allocation logging you must call the **mtrace** function. You must call the **muntrace** function to finish memory tracing. The trace information will be placed in a file specified by the **MALLOC_TRACE** environment variable.

Here's a small program to test out this system:

```
#include <malloc.h>
#include <mcheck.h>
int main()
{
    mtrace();
```



```
    char* data[10];

    for (int i = 0; i < 10; ++i) {
        data[i] = (char*)malloc(10);
    }
    for (int i = 1; i < 9; ++i) {
        free(data[i]);
    }
    muntrace();
    return (0);
}
```

This program is run with the command:

```
$ export MALLOC_TRACE=mtrace.log
$ ./leak
```

The result is a file giving us a history of our heap allocations and frees:

```
= Start
@ leak:[0x809b07c] + 0x80bb3b0 0xa
@ leak:[0x809b07c] + 0x80bb3c0 0xa
@ leak:[0x809b07c] + 0x80bb3d0 0xa
@ leak:[0x809b07c] + 0x80bb3e0 0xa
@ leak:[0x809b07c] + 0x80bb3f0 0xa
@ leak:[0x809b07c] + 0x80bb400 0xa
@ leak:[0x809b07c] + 0x80bb410 0xa
@ leak:[0x809b07c] + 0x80bb420 0xa
@ leak:[0x809b07c] + 0x80bb430 0xa
@ leak:[0x809b07c] + 0x80bb440 0xa
@ leak:[0x809b0a6] - 0x80bb3c0
@ leak:[0x809b0a6] - 0x80bb3d0
@ leak:[0x809b0a6] - 0x80bb3e0
@ leak:[0x809b0a6] - 0x80bb3f0
@ leak:[0x809b0a6] - 0x80bb400
@ leak:[0x809b0a6] - 0x80bb410
@ leak:[0x809b0a6] - 0x80bb420
@ leak:[0x809b0a6] - 0x80bb430
= End
```

Now all we have to do is to match up the **malloc** calls (denoted by the + lines) and the **free** calls (denoted by the – lines). Any **malloc** without a matching **free** is a memory leak.

Fortunately we don't have to hand match. The program *mtrace* (distributed with glibc) will do the work for you. All you have to do is feed it the name of your executable and log file:

```
$ mtrace leak mtrace.log

Memory not freed:
```



```
-----------------
   Address     Size     Caller
0x080bb3b0      0xa  at /hack/leak.cpp:10
0x080bb440      0xa  at /hack/leak.cpp:10
```

There is one limitation on the **mtrace** function. It doesn't work well for **new** and **delete**. (You get a beautiful trace showing every **malloc** and **free** being generated by the standard C++ library.) So be aware of it's limitations.

The memory tracking and consistency checking functions are not the only memory debugging facilities built into the glibc library. There are many ways to hook up diagnostic code with the memory functions. You can read the documentation for details.

However the ones presented in the last few hacks with find most of the common errors you are likely to find.

Hack 101: Generating a Backtrace

The Problem: The default abort function used by **mcheck_pedantic** is somewhat minimal. It prints an error message telling you that you are hosed but that's all. Basically you know that something went wrong, now guess where.

The Hack: Define your own memory error handling function and then use **backtrace** to get a call stack so you know where you came from.

First we need to tell **mcheck_pedantic** that we want to handle the error ourselves:

```
    mcheck_pedantic(bad_mem);
```

Now when a memory problem is found, the **bad_mem** function will be called. Of course we need to define the function:

```
static void bad_mem(enum mcheck_status status)
```

The **backtrace** function returns the call stack as an array of pointers. In order to use this function we must first allocate an array to hold this data then call the function:

```
    void* buffer[MAX_TRACE];

    int stack_size = backtrace(buffer,
         sizeof(buffer)/sizeof(buffer[0]));
```


This gives us a set of pointers, which is not useful when it comes to finding out where we are. It would be nice to turn this list into something printable with symbols. Fortunately there's a function to do that called **backtrace_symbols**. It returns a array of character pointers containing human readable (mostly) version of our pointers with some symbolic information.

```
    char** stack_sym = backtrace_symbols(
            buffer, stack_size);
```

Now all we have to do is print the results:

```
    std::cout << "Stack trace " << std::endl;

    for (int i = 0; i < stack_size; ++i) {
        std::cout << stack_sym[i] << std::endl;
    }
```

There's one more thing we have to do and that is abort the program. (With memory corrupt there's not a whole lot we can do at this point.) So the last line of our function is:

```
    abort();
```

The **abort** call was chosen because it creates a core dump we can analyze later.

A typical run of this program produces the results:

```
Stack trace
trace [0x809b458]
/lib/tls/libc.so.6 [0x401928f3]
trace(__libc_free+0x17) [0x804e817]
trace [0x809b510]
trace [0x809b521]
trace [0x809b52f]
trace [0x809b563]
trace(__libc_start_main+0x14c) [0x804c8dc]
trace(__gxx_personality_v0+0x51) [0x804c701]
Aborted (core dumped)
```

Unfortunately the **backtrace_symbols** call can't seem to find the symbols in our program. (We compiled with the debug option so they should show up.)

There are number of ways around this problem. The simplest is to run the debugger on the executable and the core file and use it to display the stack trace. This gives us the ability not only to look at the stack trace, but also the source code and the value of the variables at the time of the core dump.


```
$ gdb trace core.31957
GNU gdb 6.3-3.1.1O2mdk (Mandrakelinux)
# ... usual startup chatter ...

Core was generated by `./trace'.
Program terminated with signal 6, Aborted.

#0  0xffffe41O in ?? ()
(gdb) where
#0  0xffffe41O in ?? ()
#1  0xbffff08c in ?? ()
#2  0x00000006 in ?? ()
#3  0x00007cd5 in ?? ()
#4  0x0808118e in raise (sig=6) at ..linux/raise.c:67
#5  0x08057ca7 in abort () at abort.c:88
#6  0x0809b4e5 in bad_mem (status=MCHECK_TAIL) at
trace.cpp:28
#7  0x401928f3 in mcheck_check_all () from
/lib/tls/libc.so.6
#8  0x0804e817 in __libc_free (mem=0x6) at malloc.c:3505
#9  0x0809b510 in mem_problem () at trace.cpp:36
#10 0x0809b521 in do_part1 () at trace.cpp:40
#11 0x0809b52f in do_all () at trace.cpp:43
#12 0x0809b563 in main () at trace.cpp:49
(gdb)
```

Now we can use the normal *gdb* commands to examine the program. For example, let's take a look at the function where the error was discovered.

```
(gdb) list trace.cpp:36
31   void mem_problem()
32   {
33       char* ptr = (char*)malloc(10);
34
35       ptr[10] = 'X';
36       free(ptr);
37   }
38
39   void do_part1() {
40       mem_problem();
(gdb)
```

In this case the error is blindingly obvious and we require no further investigation. In the real world, things might no be so simple and we might have to spend more than five seconds to find the problem. But at least we have to tools in place to detect the problem early and analyze what the program is doing.

Another way of finding out where things are in the program is the *addr2line* command (part of the *glibc* package). It translates an address into a line number in the program. For example:

```
$ addr2line -C -f -e trace 0x809b510
mem_problem()
/home/sdo/hack/trace.cpp:36
```

The **-C** flag tells *addr2line* to demangle C++ symbols. The **-f** option causes the function name as well as the line to be listed. The program file is specified with the **-e** option and finally we have the address to be decoded.

This function is useful when it's impractical to use *gdb* for example when problems happen in the field and the customer doesn't want to send you a big core file, but is willing to send you the backtrace output.

Hackers know that there's more than one way to solve a problem. Sometimes we have to use the all in order to find the one what works, but fixing things in a poor working environment, with bad tools, incomplete information, and lots of pressure is something hackers do every day.

Chapter 10: Anti-Hacks

There are smart people and there are stupid people. And then there are stupid people who think they are smart. It is this type of people who can do some real damage, especially when they think they are hackers.

One of my early encounters with one of these people occurred when I had to take over from one of them and enhance a configuration program. He had designed this think with a very clever top level UI based on a very clever second abstract UI based on another clever layer of something, and so on for about eight levels of complexity. All undocumented of course.

I had a rather interesting conversation with the boss where I told him that it would take me ten weeks to modify the program, but only three weeks to write a new program that did everything the old one did and included the enhancements he waited. The new program would lack the eight level of abstraction, but it would get the job done.

A good hack is clear and easy to understand. Simple is always better unless you're Rube Goldberg and make a living off of doing things the hard way.

This chapter is devoted to the hacks that are not really hacks. By using these techniques the people who produced them show us that they know enough not to be clever, but to be really, really stupid.

Anti-Hack 102: Using "#define extern" for Variable Declarations

The Problem: In order to define a global variable you have to create an extern declaration in the header file and a real declaration in the module.

For example, a typical header (*debug.h)*:

```
extern int debug_level;
```

and a code file (*debug.cpp*):

```
int debug_level;
```

Similar declarations have to be put in two places. Is there a way to eliminate this duplication?

The Anti-Hack: Through clever use of the pre-processor you can make one header file do both declarations.

You start by creating a *debug.h* file that contains the variable declarations. Any program which uses the debug module should include the header file using the statement:

```
// Normal program file such as main.cpp
// ...
#include “debug.h”
```

The module *debug.cpp* however uses a clever pre-processor trick:

```
// File debug.cpp
// ...
#define extern /* */
#include “debug.h”
#undef extern

// -- DO NOT PROGRAM LIKE THIS !!! --
```

The **#define** statement make the **extern** keyword disappear. So now the line:

```
extern int debug_level;
```

becomes:

```
/* */ int debug_level;
```

thus you've made one file do double duty saving a lot of typing.

There are a few major problems with this code.

First you've redefined a standard C++ keyword. Now when someone looks at a file they must decide if the **extern** keyword really means **extern** or does it mean something else. C++ is bad enough when coming to you unedited. Making keywords mean one thing at one point in the program and another thing later on makes the program ten times harder to debug.

Now suppose an unsuspecting programmer comes across the *debug.h* file and decided that he wants to add a function to it which uses I/O streams. For example:

```
extern void set_output_file(std::ostream& out_file);
```

Our *debug.h file* now looks like:

```
// debug.h (incomplete)
extern int debug_level;
extern void set_output_file(std::ostream& out_file);
```


Because we are good programmers we want to follow the rules that every header file should bring any header file that it needs to do its work. In this case we reference `std::ostream` so we need the *fstream* header file:

```
// debug.h
#include <fstream>
extern int debug_level;
extern void set_output_file(std::ostream& out_file);
```

Guess what. All the **extern** declarations in *fstream* have been edited and their **extern** removed. The program is now broken. So this hack *prevents* the programmer from following good coding guidelines and making a header file self sufficient.

It gets worse. A good programmer would look at *debug.h* and see that

```
#include <fstream>
```

is missing. Of course being a good programmer he wants to make things right so he adds this line. (Not knowing of course that the *debug.cpp* contains the **extern** anti-hack.) Now when this occurs the good programmer by following good programming practices had just broken the program in strange and mysterious ways. Suddenly when he links in the debug module he gets duplicate defined symbols for thing in the *fstream* module.

A close inspection of the source code will not find the program. The *fstream* header file is a system file. It's supposed to be correct. Besides if there is a problem in it, the problem will show up in lots of programs. So it's "obvious" that the problem is not *fstream*. But it is, sort of. Because the **#define** redefined the C++ language itself, *fstream* has been rewritten and now contains errors.

This anti-hack breaks working code in a way that is difficult to locate. Do not use it.

There are a number of other problems with this hack: it doesn't allow you to locate variable definitions near the code that uses them, it doesn't let you initialize global variables, and it greatly limits what you can put in a header file. But there are minor when compared to the damage it does because it redefines the C++ language.

Anti-Hack 103: Use , (comma) to join statements

The Problem: You want your code to be as compact as possible and curly braces use too many characters. There must be a way of making your code more compact.

The Anti-Hack: Use , (comma) to join statements. For example, instead of writing:

```
if (flag) {
   do_part_a();
   a_done = true;
}
```

you can do the same thing without the curly braces:

```
if (flag)
    do_part_a(), a_done = true;
```

This saves us two lines and gets rid of two character (the curly braces).

You can get even more creative with **for** loops:

```
for (cur_ptr = first, count = 0; cur_ptr != NULL;
    cur_ptr = cur_ptr->next, ++count);
```

The **for** loop we've just defined counts the number of items in a linked list and through creative use of the , (comma) operator we've eliminated any code before the **for** and inside the body of the **for**.

As you can see the , (comma) operator is a very powerful tool for making your code very compact. But why would you ever want to do such a thing? Compact code may have been useful in the first days of computing when storage cost several dollars a byte, but today you can pick up a 128M USB Thumb drive at a convenience store.

Making the code more compact obscures the logic and make the program more difficult to read. Thus such code is more prone to errors and the errors are more difficult to find when they occur.

When you use this hack you do not show how clever a programmer you are. Experienced hackers already know about this trick. They are just mature enough not to use it.

Anti-Hack 104: if (strcmp(a,b))

The Problem: You don't want to put “unnecessary” operators in your code, especially when using **`strcmp`**.

The Anti-hack: Use the return value of **`strcmp`** bare. For example:

```
if (strcmp(name, key_name)) {
    do_special_stuff();
}
```


The logic of this code is simple and easy to understand – and extremely misleading.

Question: Is the function `do_special_stuff` executed when `name` is a `key_name` or non-key people? Remember **`strcmp`** returns 0 when the strings are *equal*. So you're only special if you are not the key person.

The code is misleading because most similar functions will return a true / false value, the true being the positive case. In other words a if **`strcmp`** followed the rules, it would return true of equal and false for not equal.

The following code is OK:

```
if (strings_are_same(name, key_name)) {
    do_key_stuff();
}
```

In this case the logic of the program and the inititation of the programmer reading the code will match and confusion is avoided. But with

```
if (strcmp(name, key_name)) {
```

the logic of the computer clash with what anyone reading the code expects and you get confusion.

One of the goals of a good hacker should be clarity. After all how can you show the world how good a hacker you are if no one understands you work. The **strcmp** code can be better written as:

```
// strcmp return value for equal strings.
static const int STRCMP_SAME = 0;

// Check for average (non-key) users and process
// their special requests.
if (strcmp(name, key_name) != STRCMP_SAME) {
    do_special_stuff();
}
```

What makes this code good? It is easy to see from the statement:

```
if (strcmp(name, key_name) != STRCMP_SAME) {
```

that the strings are not the same. But just to make sure people reading this code understand what is going a comment has been added telling the reader in English what is going on.[19]

19 The comment could be more precise and explicit except this is a made up example and I left a great deal of design to the imagination.

Anti-Hack 105: if (ptr)

The Problem: Lots of functions return a pointer when returning a good value and **NULL** on error. We a quick way of testing for **NULL**.

The Anti-Hack: Use `if (ptr)` to test to see if a pointer is **NULL** or not.

For example:

```
char* ptr = strdup(name);
if (ptr) {
    process_ptr();
} else {
    throw(out_memory_error);
}
```

Again we have a program technique that leads to extremely compact, terse, and confusing code.

The main problem with code like this is that the pointer is not a boolean. As such using it bare in an **if** statement is meaningless.

Our goal as hackers is to create wonderful and *understandable* code. Leave out short limit understand. (Leaving words out leads to short sentences, but limits but makes it difficult to understand the sentence.)

Typing

```
    if (ptr != NULL)
```

instead of

```
    if (ptr)
```

takes a fraction of a second longer, yet can save hours or days of debugging time because it makes it easier to read your program.

As hackers we want people to understand our code, marvel at it, and then extend it. Cryptic code does not help us achieve this goal.

Anti-Hack 106: The "while ((ch = getch()) != EOF)" Hack

The Problem: You need to read in a steam of characters or perform some other similar operation.

The Anti-Hack: Use a common C++ design pattern:


```
// Don't code like this
while (ch = getch() != EOF) {
   // ... do something
}
```

There are several things wrong with this code. The first is that it that you have to be careful can make sure that the variable **ch** is not a **char** variable. It must be an **int** for the comparison to work.

The second problem is that this code does multiple operations in a single statement. As a result the code is confusing and compact. It is much better to have code verbose and clear than confusing and compact.

Finally the line:

```
while (ch = getch() != EOF) {
```

is confusing. Do you know if the assignment operator (**=**) has higher precedence than not equal (**!=**)?

For example, which of these two statement is the equivalent of our **while** loop:

```
while ((ch = getch()) != EOF) {
while (ch = (getch() != EOF)) {
```

This is one example where Hack 19 comes in handy.

This is a common, compact way of writing a **while** loop. But just because it's common does not mean it's good. A more verbose, and easier to understand way of doing the same thing is:

```
// Please code like this
while (true) {
    int ch = getch();

    if (ch == EOF)
        break;

    // .. do something
}
```

Now there are people who will tell you that this hack is useful. They point out that is is very common and because it's common people recognize and understand it. And because it's so common and understandable people write it in new code making it more common and forcing more people to recognize it and understand it.

But ultimately it's sloppy pattern and leads to other sloppy coding patterns. For that reason it should be avoided.

Anti-Hack 107: Using #define to Augment the C++ Syntax

The Problem: The C++ syntax is not as rich as it could be. For example, there's no single statement to iterate through the items of a linked list.

The Anti-Hack: Augment the C++ language by using the pre-processor to define new language elements. For example:

```
#define FOR_EACH_ITEM(list) \
    for (list_ptr cur_item = list.first; \
         cur_item =! NULL; \
         cur_item = cur_item->next)
```

The problem with these syntax additions is that you've now change the C++ language to the C++++ language. What's worse it's you're own version of the C++++ language.

It's difficult enough for someone to master C++. Adding new and undocumented features to the language makes things worse. Even if someone takes the trouble to document them the people maintaining the program have to learn a new and unique language.

Another problem is that macros like this tend to hide errors. For example the following will result in an infinite loop or a core dump:

```
FOR_EACH_ITEM(name_list)
    std::cout << cur_item->name << std::endl;
```

It's difficult to spot the error because the mistake is hidden inside the **FOR_EACH_ITEM** macro. For readers of this book the mistake is somewhat easy to spot because all you have to do to find the macro definition is to look back to the previous code example. In real life things are much worse. The macro definition is usually hidden in a header file and you have to go hunting for it.

In general it's better to write code who's syntax is visible and standard. That let's the people who come after you understand your work so that they can improve and enhance it.

Anti-Hack 108: Using BEGIN and END Instead of { and }

The Problem: The curly braces **{** and **}** are terse and cryptic. Besides they look nothing like what you see in ALGOL programming.

The Anti-Hack: Use **`#define`** to define a set of macros that allow you to make C code look like ALOGL-68.

Here's a small sample:

```
#define BEGIN {
#define END }

#define IF      if (
#define THEN    ) {
#define ELSE    } else {
#define ENDIF   }
```

The code looks like:

```
WHILE (system_ready())
    BEGIN
       int success = process_cmd();
       IF success THEN
            good();
       ELSE
            bad();
       ENDIF
    END
```

This makes things simple and easy to understand if you're an ALGOL-68 programmer, it makes things next to impossible if you're a C programmer.

Most programmers when faced with this sort of code, curse, then use the pre-processor to remove the macros and get back to the underlying C code.

This hack is of historical interest because the original Borne shell was written this way. Fortunately it was quickly replaced by a C version much to the relief of all the people who had to maintain this program.

Anti-Hack 109: Variable Argument Lists

The Problem: You have function which takes a lot of different parameters. Specifying them as traditional arguments is impractical. So what do you do?

The Anti-Hack: Use the variable argument list feature of C++ to pass in a variety of aguements.

For example, let's suppose we are drawing a box. We can specify a width, height, line width, color, and other items. All of these are optional, if we don't specify one a default is used.

Here's a typical call to our **draw_box** function:

```
    draw_box(BOX_WIDTH, 5, BOX_HEIGHT, 3, NULL);
```

In this case the NULL is used to indicate the end of the argument list.

This concept can be expanded even further. Suppose this box contains a list of words. Now we can do the following:

```
    draw_box(BOX_WIDTH, 5,
             BOX_WORDS, “alpha”, “beta”, NULL,
             BOX_HEIGHT, 3, NULL);
```

So the word list (BOX_WORDS) now takes a variable list of words as it's parameter. So we have variable parameter lists in variable parameter lists. This is extremely clever!

This is also extremely dangerous and stupid. First of all this completely defeats the C++ type checking mechanism. In this example, BOX_WIDTH is supposed to be followed by an **int**. But what if we follow it with a string.

```
    draw_box(BOX_WIDTH, “large”,
             // ....
```

The compiler has no way of knowing that the wrong type has been supplied. The runtime has no way of telling a pointer from an integer. On most systems a pointer looks like a very large pointer. So the result of this code is a very large box and a very confused programmer.

But that's not the worst. What happens if you leave out a NULL? If you leave out the terminating NULL, the function has no way of telling where the end of the argument list is. What will happen is that it will go through the arguments and look for a NULL to stop on. Not finding one it will continue on using whatever is store on the stack are arguments, until by accident it finds a NULL. Or crashes, or something else evil happens.

In any case the result is unexpected and difficult to diagnose. Because this system defeats so many of the C++ safety checks it should be avoided.

There are lots of other ways of passing in large number of arguments which provide both flexibility and safety. (For example see Hack 4.) Use one them instead.

Anti-Hack 110: Opaque Handles

The Problem: You don't want to reveal your implementation to the user. After all he doesn't need to know what's going on inside your code, only how to call it.

The Anti-Hack: Give the user opaque handles to work with. In other words cast implementation dependent information into something like a **`void*`** or **`short int`** that the user can't play with.

For example:

```
typedef short int font_handle;
typedef short int window_handle;

font_handle the_font = CreateFont(“Courier”, 10);
window_handle the_window = CreateWindow(100, 300);

DrawText(the_font, the_window, “Hello World”, 30, 30);
```

Now some people might ask “What's wrong with this?” After all the user doesn't need to know the details of what makes up a font or a window. By giving him an opaque type we hide all this information from him.

The problem is that the information is also hidden from the compiler. There is absolutely no way that the compiler can tell the different between a font handle and a window handle.

In our example the `DrawText` function takes a window handle and font handle as its first two arguments. In that order! In the same sample code above the order is switched and there is nothing the compiler can do to discover the error.

It is possible to create type checked opaque handles which which hide the implementation from the user and still allow for compile time checking. These are described in Hack 31.

Anti-Hack 111: Microsoft (Hungarian) Notation

The Problem: It's impossible to tell the type of a variable just from looking at the name. For example is `box_size` a floating point number, an integer, or a structure?

The Anti-Hack: Microsoft Notation.

Microsoft Notation (also known as Hungarian Notation[20]) is a naming system where each variable is prefixed with type information. For example integers begin with “i” and floating point variables begin with “f”. Short integers begin “si” as in `siSize`. Unless the short integer is a handle to something, then the “h” notation kicks in. For example, `hfCurrentFont` is a font handle. Of course some people believe that “h” is too cryptic so they use “hndl” as `hndlCurrentFont`, but this can lead to confusion as the same prefix can be used for font handles, window handles, and all the other many other handles you have to deal with in Microsoft windows.

There are a number of problems with this system. The first is that the whole idea of having a compiler and a high level language is to hide the details of the implementation from you. Putting this barf on the beginning of each variable gives you information that you really don't care about.

The compiler needs to know detailed type information on every variable, you do not.

The second problem is that there is no standard list of prefix characters. Many people invent their own leaving you to guess what the difference between `pszName` and `szName` is. The answer is that the first is a pointer to a string ending in a zero character (implemented by **char***) and the second is a string ending in a zero character (implemented by **char***). In other words, there is no difference.

Finally there is the problem of flexibility and program updates. Suppose you have implemented an address book and address is stored in a structure (`stAddress`). You improve the program and replace the structure with a class (`cAddress`). Are you going to go through your entire program and update the prefix of every address variable in it? If you do it's a lot of work. (Not good.) If you don't you now have variables which use an incorrect prefix. (Very not good.)

This is a loose and loose worse situation.

You should write variable names using as clear as English (or whatever your native language is) as possible. Leave type information to the compiler. Don't mess up good words with random barf. Programs are obscure enough without using notation systems which obscure them more while adding no value to the program.

20 The Hungarians are nice people and should be associated with such a bad hack. So that's why the term Microsoft Notation is used here even though the term Hungarian Notation is more commonly used.

Microsoft Notation for English

Over the years C and C++ programmers have devised many ways of making their writing clearer. As a result program are much more readable than ever before. I've often wondered if the lessons learned by programming could be applied to English as well.

Take Microsoft notation (aka. Hungarian notation) for example. It puts a prefix on each variable name telling us what the type of the variable. Wouldn't English be much clearer if we identified each verb with "v", each noun with "n" and so on.

pAfter avAll aMany nEnglish nWords vCan vHave aMore cThan nOne nMeaning. pFor nExample "mall" vIs prBoth aA nVerb cAnd aA nNoun. pWith pnOur aNew nNotation pYou vCan vWrite nSomething aLike, "aThere vWas aSuch aA nCrowd pAt arThe nMall aThat pnWe vWere vMalled" cAnd nPeople vCan aEasily vUnderstand pYou.

Chapter 11: Embedded Programming Hacks

The number of computers is exploding. The are now in phones, washing machines, televisions, toasters, and watches.[21] Programming for these non-standard platforms can be a real challenge.

I got my first programming job while I was in high school. The boss told, "You are creating the embedded software for the Camsco Waterjet I cutter."

"Great," I said. "Where's the machine?"

"You see these boxes," he replied pointing around the room. Indeed there was a moderately large stack of boxes against one one. "They contain the parts we're going to build it out of."

"Great," I said, "Where's the hardware documentation."

"We're still in the process of designing the hardware. As soon as we get the prototype done, we'll start writing the documentation."

"What about the I/O controller documentation?"

"Some of them are off the shelf, and I can get you documentation for those. But there are a couple that we are making ourselves and they are still being designed. You'll have to wait for information on those the design is not solid yet."

The job turned out to be a very interesting challenge for all involved. For my part I learned how to write very flexible code that could be easily configured to work with a specific hardware specification. That was mostly self defense as getting the hardware specification was difficult, getting it on schedule was even more difficult, and getting correct information – impossible. (See the first hack below.)

Testing The WaterJet

Testing the first WaterJet was a challenge for all involved. It was the first production water jet cutter ever made so everything was new to all of us. In order to get things right we tested and tuned the machine for over a year.

21 There is actually a watch on which you can run Linux!

We had a deal with the shoemaker who was buying the machine. He would supply us with raw material for testing and we would give him back any pieces cut out during the test process so he could make tennis shoes out of them – or so we thought.

In order to get consistent timing results we did all our benchmarking using a single size – *9 right*. So for about 5-7 months we did nothing but cut out a batch of 9 rights, tune the machine, and ship the cut pieces to the shoemaker. This was a very large company with many people who turn out hundreds of thousands of shows each year.

After finally getting the machine tuned and working really well, we started the process of breaking it down for boxing and shipping to the customer.

It was then that we got a call from the factory foreman. "Are you the people who keep shipping us these 9 rights?"

We told him we were.

"Finally I tracked you people down. Purchasing had no idea who you were because we never ordered any 9 rights from anyone. I never thought to look at our capitol equipment acquisition group until now."

"Is there some problem with our cut pieces?" We were a little worried at this point.

"Not exactly, but do you realize that you sent us 10,000 nine-rights and no lefts!"[22]

Hack 112: Always Verify the Hardware Specification

The Problem: Hardware specifications are frequently incomplete, incoherent, incomprehensible, out of date, or missing.

While I was working on the Camsco WaterJet, I had to program a status panel. This device contained 10 warning lights. I got the handwritten specification from the hardware guy and decide to test things.

22 His existing process used a die to cut out a left and right at the same time and he had on way of cutting out just 9 left.

Light 1 was supposed to be “Oil Pressure Low” and it was “Ram failure”. Light 2 was supposed to be “Water Supply Failure” and it was “Positioner Error.” Light 3, “DC Power Failure” turned out to be “Compressor Failure”.

So I wrote up a nice document describing where how the system was really wired and I handed this to the hardware guide telling him “Woody, the lights are all mixed up.” I handed him the paper, “Here's how they are currently wired up.”

He took the paper from my hand and went straight to the copy machine and made a copy. He then handed me the copy (not the original) and said, “Here's the new specification.”

The Hack: Test early, test often.

The first thing you should do when faced with any new piece of hardware is to create a short diagnostic program that tests out it's basic functionality. It is this program which lets you discover things like:

- Commands that don't function as documented.
- Byte order problems (which of course are not documented).
- Document that contains non-standard bit order notation. (I've actually read one hardware manual that put the least signification bit on the left and the most significant on the right.)
- Other omissions, errors, ambiguities, and problems with the documentation.

Dealing with new hardware for the first time is a difficult and challenging operation. Far too many hardware people don't speak software and don't know how to write decent documentation. (A lot of software people are just as bad when it comes to writing things down.)

By testing early you can find out how the hardware really works instead of how it's supposed to work. That way you can write a program that really does the job instead of one that supposed to do the job.

Hack 113: Use Portable Types Which Specify Exactly How Wide Your Integers Are

The Problem: The C++ does not define how many bits are in the **int** type. (Or any other type for that matter.) In embedded systems we need to know the exact number of bits in order to talk to the hardware.

The Hack: Define portable types or use the portable types already defined for you.

The header file **<sys/types.h>** contains a set of type definitions provide exact definitions for integer values. The include:

int8_t	**int16_t**	**int32_t**	**int64_t**
u_int8_t	**u_int16_t**	**u_int32_t**	**u_int64_t**

By using these types you are guaranteed that the integers you define have the exact number of bits you expect.

If your system doesn't have these types in a header file somewhere, then define your own. (And of course verify them before you publish them.)

Hack 114: Verify Structure Sizes

The Problem: The compiler's idea of the size of a structure may be different from your idea.

Here's a structure definition for the old RIO MP3 player. Because the structure must exactly match the hardware specification, the offset of each field is included in the comments. The other thing to note is that this block must be exactly 512 bytes long (again, this number came from the hardware specification.)

```
// directory header
struct rio_dir_header
{
    u_int16_t entry_count;       //[0] Number of entries
    u_int16_t blocks_available;  //[2] Blocks available
    u_int16_t Count32KBlockUsed; //[4] Blocks used
    u_int16_t block_remaining;   //[6] Blocks remaining?
    u_int16_t Count32KBlockBad;  //[8] Bad blocks
    int32_t TimeLastUpdate;      //[10] Last time updated
    u_int16_t check_sum1;        //[14] Checksum part 1
    u_int16_t check_sum2;        //[16] Checksum part 2
    char not_used_1[2];          //[18] Junk
    u_int16_t Version;           //[20] Version number
    char not_used_2[512 - 22];   //[22] Not used
};
```


When music was loaded into the RIO player some problems were observed. First each time you started to play a song, you got a short blast of noise followed by the song. Secondly if you pressed the fast forward button, the player would jump to the next song.

So what is going on here? This problem is especially difficult to debug because there's no official documentation on the RIO's interface and there's absolutely no way of telling what's going inside the device.

However, you may have noticed that we only included the header definition in this book, so that may give you some idea. The actual problem concerns the way that the compiler lays out structures.

We can discover through testing what the real offsets of each field are. The results are slightly different from the ones we hand computed and put in the comments:

```
u_int16_t block_remaining;   //[6] Blocks remaining?
u_int16_t Count32KBlockBad;  //[8] Bad blocks
int32_t TimeLastUpdate;      //[10 12] Last time updated
u_int16_t check_sum1;        //[14 16] Checksum part 1
```

So how come the two byte `Count32KBlockBad` takes up four bytes? The answer is that the compiler wants to align 32 bit values (`TimeLastUpdate`) on a 4 byte boundary. The offset 10 is not on a 4 byte boundary, so the compiler puts in a couple of padding bytes to move the field to the next four byte boundary.

So the real structure layout is:

```
u_int16_t block_remaining;  //[6] Blocks remaining?
u_int16_t Count32KBlockBad; //[8] Bad blocks
u_char  hidden_pad[2];      //[10] Added by compiler
int32 TimeLastUpdate;       //[10 12] Last time updated
u_int16_t check_sum1;       //[14 16] Checksum part 1
```

So our 512 byte structure is really 514 bytes and some of the fields are in the wrong place.

(Some of you might ask the question “If **`check_sum1`** and **`check_sum2`** are in the wrong place, why doesn't the RIO detect that the checksums are incorrect and display an error message? Good question, but as a software hacker you should know better than to expect consistency, sanity, or common sense from a hardware designer.)

Now that we know what caused this problem what can we do to prevent things like this from occurring in the future?

The Hack: Verify the computed size of structure against the actual size.

Simply put, our program needs the following line:

```
assert(sizeof(struct rio_dir_header) == 512);
```

This should be done for every data structure shared by the software and the hardware. As we have seen the software's idea of what goes on in a structure can be different from the hardware's.

A possible cure for this problem can be found in Hack 116.

Hack 115: Verify Offsets When Defining the Hardware Interface

The Problem: The compiler can not only pad structures but also move fields around.

The Hack: Verify the offset of every field in a structure using the **offsetof** operator.

Back to our previous example, let's verify not only the size of the structure but the location of each field as well:

```
assert(sizeof(struct rio_dir_header) == 512);

assert(offsetof(struct rio_dir_header,
                entry_count) == 0);

assert(offsetof(struct rio_dir_header,
                blocks_available) = 2);

// ...
```

Hack 116: Pack Structures To Eliminate Hidden Padding

The Problem: The compiler pads things without our permission. This screws things when we are dealing with hardware.

The Hack: Use the *gcc* packed attribute to tell *gcc* to make the structure as compact as possible.

For example:

```
struct rio_dir_header
{
    u_int16_t entry_count;       //[0] Number of entries
// ....
} __attribute__((packed)); // Hack 41
```

This is an example of the hacker using the features provided by the compiler to the fullest. If you are not using *gcc* read your compiler's documentation. Chances are they have something similar to the packed attribute.

If your compiler doesn't support packing of any type, you'll have to split the long field `TimeLastUpdate` into smaller two **`u_int16_t`** fields that the compiler will pack.

When encountering a problem like this a good hacker will find some way of going over, under, or through roadblocks put up by the compiler.

Hack 117: Understand What the Keyword volatile Does and How to Use It.

The Problem: A good compiler will remove useless code. In embedded programming some code is not as useless as it appears.

Let's suppose we are dealing with a serial I/O controller. In order to clear the device we need to unload the three character input buffer. The code for this is:

```
// Won't work

// The device is a memory mapped I/O device
// Initialize the pointer to the input register
char* serial_char_in = DEVICE_PTR_SERIAL_IN;

// ...
char junk_ch;   // A junk character
junk_ch = *serial_char_in;  // Read one useless character
junk_ch = *serial_char_in;  // Clear second useless char
junk_ch = *serial_char_in;  // Third character gone
                            // buffer now clear
```

So what does this code really do? It assigns `junk_ch` the top character from the device. This is repeated three times so clear the three character buffer.

But the compiler is smart. It looks at this code and says, "That first assignment is useless. We compute the value of `junk_ch` only to throw it away. I can eliminate that line and the program will act the same."

So the optimizer rewrites the code to look like:

```
char junk_ch;   // A junk character
junk_ch = *serial_char_in;  // Read one useless character
junk_ch = *serial_char_in;  // Clear second useless char
junk_ch = *serial_char_in;  // Third character gone
```

Now it looks at the second line. It too can be eliminated.

The third line can only be eliminated if `junk_ch` is not used somewhere later in the program. In this example, it's not so it too is eliminated. The resulting code is very nicely optimized:

```
char junk_ch;   // A junk character
junk_ch = *serial_char_in;  // Read one useless character
junk_ch = *serial_char_in;  // Clear second useless char
junk_ch = *serial_char_in;  // Third character gone
```

And since `junk_ch` is now never used, the compiler can make the variable disappear and save the stack space it would use.

So the optimizer had saved us time by cutting out useless instructions and memory by eliminating a useless variable. What's wrong what that?

The answer is that the program no longer functions. The device is no longer getting cleared. This is one problem with embedded programming, the compiler does not know about the strange side effects that can occur when dealing with memory mapped I/O.

The Hack: Use the **volatile** keyword to identify memory mapped device and other **volatile** data.

The **volatile** keyword tells C++ that a variable may be changed at any time by forces outside the control of the normal programming environment. More specifically it tells the compiler that optimizer is not allowed to change the number or order of any access to this variable.

```
volatile char* serial_char_in = DEVICE_PTR_SERIAL_IN;
```

Now we we execute:


```
junk_ch = *serial_char_in;  // Read one useless character
junk_ch = *serial_char_in;  // Clear second useless char
junk_ch = *serial_char_in;  // Third character gone
```

the I/O device will be accessed three times and we will clear the device.

Alert readers may notice that we violated Hack 3 and didn't use **const** where we should have. Since we don't what to change the pointer, it should be made **const** giving us:

```
volatile char* const serial_char_in =
        DEVICE_PTR_SERIAL_IN;
```

Now we have enough advanced keywords in a single variable declaration to identify us as a real hacker.

Hack 118: Understand What the Optimizer Can Do To You

The Problem: Early in my career I had to work with a slow and poorly designed disk controller. One of the things it did wrong as to generate an interrupt in the following manner:

1. Trigger the interrupt line

2. Set the interrupt bit in the status registering telling the system why the interrupt occurred.

There was a problem with this system. I was working on a fast processor this was a slow disk controller. As such the entire interrupt service routine could complete it's work between the time that the controller performed step #1 and step #2.

Until I figured this out my code was constantly complaining of spurious interrupts and lost interrupts.

The problem would start the interrupt, I would read the register (nothing set – log a spurious interrupt), the interrupt routine would finish, then the controller would set the status bit indicating what I/O was available. (Of course by this time I'd dismissed the interrupt, so the interrupt for that I/O was lost.)

The “solution” to this problem was to add a delay loop to the code at the beginning of the interrupt service routine to give the I/O controller time to get caught up with events.

```
/*
 * WARNING: This is highly processor dependent
 */
```



```
const int DELAY_COUNT = 100; // Loop this many times

int result;

for (int i = 0; i < DELAY_COUNT; ++i) {
    result = 12 * 34;
}
```

But there are some surprises in this code. First of all how many times is the executed:

a) Zero times

b) One time

c) 100 times

If you guessed “a” you're right – sometimes. But “b” and “c” are also right – sometimes.

The problem is that the optimizer may play all sorts of games with this code. The optimizer may look at the code and determine that all this code is waste time so it can be eliminated. So the loop will execute zero times.

If **`result`** is used later in the code, then one trip through the loop is enough and the optimizer can drop the other 99.

The GNU C++ compiler is smart. Its optimizer says “This code looks like the useless code you're find in a delay loop, so I'll generate the code to go through the loop 100 times.”

The second question is “How many times is the multiply done?” Even without the optimizer the answer to this question zero. Any modern compiler knows how to do constant folding and will evaluate constant expressions at compile time so they don't have to be computed at run time.

The Hacks: Actually there are two hacks in play here. The first is to periodically look at the assembly code to see what the compiler is doing to you. The second is to use the **volatile** modifier to force the compiler to not optimize your code.

Let's take a look at what the GNU compiler does with our original delay loop:

```
    movl $99, %eax
.L5:
    decl %eax
```



```
    jns  .L5
```

So we have the loop part, but not the multiple part. This will generate some delay but not enough.

So let's rewrite the loop using **volatile** variable. The **volatile** keyword tells the compiler that this variable is special and to make no assumptions about the saneness of this variable.

Here's the new, improved loop:

```
    volatile int result;
    volatile int f1 = 12, f2 = 34;

    for (int i = 0; i < 100; ++i) {
        result = f1 * f2;
    }
```

Now from the assembly code we can see that the multiply is actually happing and our delay loop is causing a delay.

```
    movl $99, %ecx
    movl $12, -8(%ebp)
    movl $34, -12(%ebp)
.L5:
    movl -8(%ebp), %eax
    movl -12(%ebp), %edx
    imull %edx, %eax
    decl %ecx
    movl %eax, -4(%ebp)
    jns  .L5
```

Finally a good hacker would design a short test program to test the length of this delay loop. Delay loops are extremely difficult to get right and the processor might have some surprises in store for you as well.

It should be pointed out that hackers avoid writing delay loops like this as much as possible. Such loops are dangerous to code, non-portable, and a waste of CPU resources. They should only be created if there is there is no other way of getting the program to function. But now at least if we're forced to use them, we can get them right.

Hack 119: In Embedded Programs, Try To Handle Errors Without Stopping

The Problem: Embedded programs run in their own environment. Often it's difficult if not impossible for the user to start and stop services. Errors which cause a program to stop or crash can easily take down the entire device.

If you need a reminder about how much damage bad error handling can damage a system see the sidebar *A Real System Crash* on page 44.

The Hack: Never stop. And even if you do stop, start right back up again.

Let's take a typical programming example where a client must connect to a server:

```
ConnectionType *the_connect =
    new ConnectionType(CONNECTION_PARAMETERS);
```

The question is what happens when the server is not running? In the case of a typical non-embedded application, you bail out with an error message. Basically you tell the user “You've got a problem, fix it and then run me again.”

```
// Typical non-embedded code
the_connection->connect_to_server();
if (the_connection.bad()) {
    std::cerr <<
        “FATAL ERROR: Could not connect to server “ <<
         std::endl;
    abort();
}
```

Note: In this example, the `ConnectionType` class sets an error flag instead of throwing an exception. Programming wise exceptions are a cleaner way of doing things. But they clutter up an example with extra statements, so writing wise flags are clearer. But you should use exceptions in real programming.

Now in embedded programming the user can't stop and start programs. So if a vital program dies, the machine becomes useless to him. There's even a term for this, verb: “brick” as in “When he tried to use the wireless function he bricked the machine.”

Since we don't want our systems to dies like this, we need to modify our programming methodology and if possible, never stop. Here's how an embedded hacker would deal with connecting to the server:


```
// Good embedded code
while (1) {
    the_connection->connect_to_server();

    if (! (the_connection.bad())
        break;

    log_warning(
        “Unable connect. Will retry in %d seconds”,
        CONNECT_DELAY);
    sleep(CONNECT_DELAY);
}
```

So what happens if the server is not ready? In this case we log an error (to tell someone we had a problem), then wait a few seconds (no need to pound the server), and try again.

So even if it takes a while for the server to come up, we will eventually connect to it. If the server never comes up, the we never come up, but that is not our problem. It is up to whoever is handling the server to figure out a way for it to come up, and when it does we're ready.

So the basic rule is that the program will keep trying until it succeeds. It never stops because it's almost impossible to start it again. (In badly programmed systems, the only way to restart dead processes is to power cycle the machine.)

But what happens if your program encounters a problem where retrying is not an option. For example:

```
try {
    char *buffer_we_must_have = new char[BUFFER_SIZE];
    // ....
}
catch (std::bad_alloc e) {
    // ?? What do we do now?
}
```

There is no way we can keep retrying the **new** operation until we get enough memory. After all do we expect the user to solve the problem by going out, buying a memory chip and installing it while our program keeps running?

No, in this case we must abort. The program can no longer function. But in embedded programming we should keep things running whenever possible. So one hack around the abort problem is to make sure that if a program does abort, we start it right up again.

So we create a small executive program which monitors our main program:

```
while (1) {
    system(“main_program”);
    log_warning(“main_program stopped.  Will restart.”);
    sleep(SETTLE_TIME);
}
```

As hackers we always know that there is some way around “impossible” restrictions. In embedded systems, we can't stop, and if we do stop, we don't stop for long. No matter what happens, the system must run.

Hack 120: Detecting Starvation

The Problem: In an embedded system it's possible for a high priority process to get stuck and starve all the lower priority processes of CPU power. How do we detect such a problem.

The Hack: Use the combination of a watchdog and a low priority process to keep the system running.

A watchdog is a device designed to help keep a system running. After you set it up it sits there and waits for you to ping it every so often. If for some reason you don't contact it in a specified time period it will reset your system.

Through the use of a watchdog and a low priority task to tickle the watchdog you can make sure that if you system has a problem and locks up, that it will then reset and continue.

Here's a typical program designed to service a watchdog timer:

```
// If we don't do anything for 5 minutes, reboot
const int WATCHDOG_TIME = 60 * 5;

// Tickle the watchdog every 1/2 minute
const int SLEEP_TIME = 30;

int main()
{
    init_watchdog();
    set_my_priority(LOWEST_POSSIBLE_PRIORITY);

    while (true) {
        tickle_watchdog();
        sleep(SLEEP_TIME);
    }
```



```
}
```

The idea is that watchdog expects to have someone tickle it every five minutes. If it doesn't it will get angry and reboot the system.

We are going to tickle is every thirty seconds. (That's much less that five minutes) so we should be safe.

If for some reason the system gets really busy, so busy it can't give our poor tickle process any CPU in five minutes, then the program will fail to tickle the watchdog, the watchdog will get upset and we'll reboot.

There are a number of hardware based watchdog boards available as well as a software only kernel driver. These are all documented in the kernel source in the directory */usr/src/linux/Documentation/watchdog*.

It should be noted that this hack is a solution of last resort. It should only be used where the choices are to let the system be permanently hung up or reset the system. Hopefully the other protections you've got built into the system will prevent you from ever having to let the watchdog do its job. But it a good failsafe against the worst happening.

Chapter 12: Vim editing hacks

Vim is a high powered editor based on the old UNIX *Vi* program. However, it has tremendously extended features beyond the *Vi* command set.

Because *Vim* was made by programmers for programmers. As such it contains a lot of commands designed to make a programmer's life easier. This chapter discusses the major ones which include:

- Syntax Coloring
- Using *Vim*'s built-in make system.
- Automatic Indentation
- Source code exploration
- Viewing the Logic of Large Functions
- Log file viewing

Even if you are programming on Microsoft Windows you can use *Vim*. Although programming environments like Visual C++ provide you with an editor, the *Vim* is far superior at editing that their internal editor.

Note: This chapter assumes that you have turned on *Vim* extended features. On UNIX and Linux systems this is done by creating a *~/.vimrc* file. On Microsoft Windows this is done by default.

What is *Vim*

Vim is a high quality text editor who's one job is to edit text fast. It is designed for people, like programmers, who have to edit a lot of text.

One of the problems with *Vim* is its steep learning curve. The main cursor movement keys are **h**, **j**, **k**, and **l**. If you look on a keyboard you'll see that these keys are the “home” type positions on the right hand. In other words, they are the fastest and easiest keys for most people to type.

But they are totally non-mnemonic. (For example, right is **l** (lower case L).) Learning *Vim* takes time, but one you do learn them you can edit quicker than almost any other editor.[23]

Hack 121: Turning on Syntax Coloring

The Problem: Text editing using a simple monospaced font with no color.

The Hack: Turn on *Vim*'s syntax coloring.

To turn on syntax coloring in *Vim* simply execute the command:

```
:syntax on
```

Now things such as keywords, strings, comments, and other syntax elements will have different colors. (If you have a black-and-white terminal, they will have different attributes such as bold, underline, blink, and so on.)

Actually you usually don't execute this command manually. Instead it's put in your *$HOME/.vimrc* file so that it is automatically executed each time *Vim* is started.

Hack 122: Using Vim's internal make system

The Problem: You have to exit *Vim* to build a program, then go back into *Vim* to fix the problems.

The Hack: Use *Vim*'s `:make` command to perform the builds.

The *Vim* `:make` command builds the program by running the *make* program. (On Windows the *nmake* command is used.) To start the build process execute the command:

```
:make [arguments]
```

If errors were generated, they are captured and the editor positions you where the first error occurred.

Take a look at a typical **`:make`** session. (Typical **`:make`** sessions generate far more errors and fewer stupid ones.) Figure 6 shows the results. From this you can see that you have errors in two files, *main.c* and *sub.c*.

23 *Emacs* users will probably challenge this statement as well as people with other favorite editors. If you are one of these people you can easily skip this chapter.


```
:!make | & tee /tmp/vim215953.err
gcc -g -Wall -o prog main.c sub.c
main.c: In function `main':
main.c:6: too many arguments to function `do_sub'
main.c: At top level:
main.c:1O: parse error before `]'
sub.c: In function `sub':
sub.c:3: `j' undeclared (first use in this function)
sub.c:3: (Each undeclared identifier is reported only once
sub.c:3: for each function it appears in.)
sub.c:4: parse error before `]'
sub.c:4: warning: control reaches end of non-void function
make: *** [prog] Error 1
2 returned
"main.c" 11L, 111C
(3 of 12): too many arguments to function `do_sub'
Press RETURN or enter command to continue
```

*Figure 6: **:make** output.*

When you press Enter (what *Vim* calls Return), you see the results shown in Figure 7.

```
int main()
{
        int i=3
        do_sub("foo");
        ++i;
        return (0);
}
}
~
(3 of 12): too many arguments to function do_sub
```

Figure 7: The first error.

The editor has moved you to the first error. This is line 6 of *main.c*. You did not need to specify the file or the line number, *Vim* knew where to go automatically. The following command goes to where the next error occurs (see Figure 8):

```
:cnext
```



```
int main()
{
       int 1=3
       do_sub("foo");
       ++i;
       return (0);
}
}
~
(5 of 12): parse error before `}´
```

Figure 8: ***:cnext****.*

The command **:cprevious** or **:cNext** goes to the previous error. Similarly, the command **:clast** goes to the last error and **:crewind** goes to the first. The **:cnfile** goes to first error message for the next file (see Figure 9).

```
int sub(int i)
{
       return (i * j)
}
~
~
~
~
~
~
(7 of 12): `j' undeclared (first use in this function)
```

Figure 9: ***:cnfile*** *command*

If you forget what the current error is, you can display it using the following command:

```
:cc
```

If you have already run make and generated your own error file, you can tell *Vim* about it by using the **:cfile *error-file*** command. Where ***error-file*** is the name of the output of the make command or compiler. If the ***error-file*** is not specified, the file specified by the **'errorfile'** option is used.

Another command which may be of use to you is the **:copen** command. It opens a new window containing a list of errors and lets you select the error you want to examine by moving to it and pressing **<enter>**.

Finally the **`:help`** command can be used to get documentation on all the commands presented here.

Hack 123: Automatically Indenting Code

The Problem: Indenting C++ programs is a chore.

The Hack: Let *Vim* do it automatically.

The *Vim* editor contains a very smart indentation system. To access it all you have to do is turn it on with the command:

```
:set cindent
```

By default the indentation size is 8 spaces. If you wish something different, like for example an indentation size of 4, you need to execute this command:

```
:set shiftwidth=4
```

Again, these commands are frequently put in the *$HOME/.vimrc* file so they are executed every time *Vim* starts.

Now when you type code as you press enter, the program will be indented automatically. Figure 10 illustrates how **`'cindent'`** works.

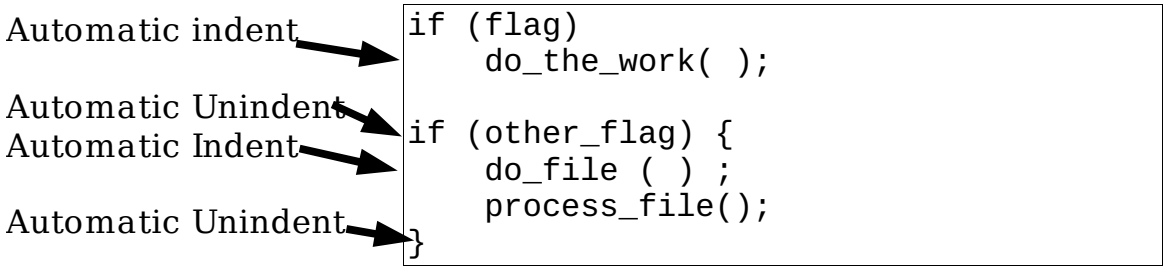


Figure 10: ***`cindent`***.

Note: You may not want to turn on **`'cindent'`** for all types of files. It would really mess up a text file. The following commands, when put in a *.vimrc* file will turn on **`'cindent'`** for C and C++ files only.

```
:filetype on
:autocmd FileType c,cpp :set cindent
```

Hack 124: Indenting Existing Blocks of Code

The Problem: You are give a legacy program which has been edited so many times so as to make the indentation useless.

The Hack: Use *Vim*'s internal indentation functionality to do the work for you.

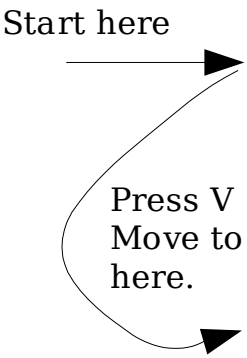
To indent a block of text using *Vim* execute the following commands:

1. Position the cursor on the first line to be indented.

2. Execute the **V** (upper case “v”) command to start “VISUAL LINE” mode.

3. Move to the last line to be indented using any of the *Vim* movement commands. The block to be indented will be highlighted. (See Figure 11.)

4. Execute the *Vim* command **=** to indent the highlighted text.




```
// Code intentionally indented wrong
int do_it()
{
    if (today)
        do_today();
        do_now();
    do_required();
do_finish();
}

int main()
{
   do_it();
   return(0);
}
```

Press **=**. Result:

```
// Code intentionally indented wrong
int do_it()
{
    if (today)
        do_today();
    do_now();
    do_required();
    do_finish();
}

int main()
{
   do_it();
   return(0);
}
```

Figure 11: Indenting Code.

Hack 125: Use tags to Navigate the Code

The Problem: You've working on a bunch of code you're not familiar with and wish to go through the code step by step. You've come to a call to `do_the_funny_bird` and you want to see what this function does, but you don't know where it's defined. How do navigate through the code effective in such cases.

The Hack: Use the *ctags*[24] command to generate a location file (called *tags*) which *Vim* can use to locate function definitions.

You need to generate the *tags* file before you start editing. This is done with the command

```
$ ctags *.cpp *.h
```

Now when you are in *Vim* and you want to go to a function definition, you can jump to it by using the following command:

```
:tag do_the_funny_bird
```

This command will find the function even if it is another file. The **CTRL-]** command jumps to the tag of the word that is under the cursor. This makes it easy to explore a tangle of C++ code.

Suppose, for example, that you are in the function `write_block`. You can see that it calls `write_line`. But what does `write_line` do? By putting the cursor on the call to `write_line` and typing **CTRL-]**, you jump to the definition of this function (see Figure 12). The `write_line` function calls `write_char`. You need to figure out what it does. So you position the cursor over the call to `write_char` and press **CTRL-]**. Now you are at the definition of `write_char` (see Figure 13).

24 Make sure you get the extended version from http://ctags.sourceforge.net/


```
void write_block(char line_set[])
{
    int i;
    for (i = 0; i < N_LINES; ++i)
        write_line(line_set[i]);
}
```

CTRL-] goes to the definition of `write_line` (switching files if needed).

The command **`:tag write_line`** does the same thing

```
void write_line(char line[])       CT
{                                  of w
      int i;                       if n
      for (i = 0; line[0] != '\0')
             write_char(line[i]);
}
~
"write_line.c" 6L,
```

*Figure 12: Tag jumping with **CTRL-]**.*

```
void write_char(char ch)
{
    write_raw(ch);
}
~
"write_char.c" 4L, 48C
```

CTRL-] with cursor on `write_char` gets us here

Figure 13: Jumping to the `write_char` tag.

The **`:tags`** command shows the list of the tags that you have traversed through (see Figure 14).


```
~
:tags
    # TO tag             FROM line   in file/text
    1  1 write_block      1           write_block.c
    2  1 write_line       5           write_block.c
    3  1 write_char       5           write_line.c
>
Press RETURN or enter command to continue
```

Figure 14: The ***:tags*** *command.*

Now to go back. The **CTRL-T** command goes the preceding tag. This command takes a count argument that indicates how many tags to jump back.

So, you have gone forward, and now back. Let's go forward again. The following command goes to the tag on the list:

```
:tag
```

You can prefix it with a count and jump forward that many tags. For example:

```
:3tag
```

Figure 15 illustrates the various types of tag navigation.

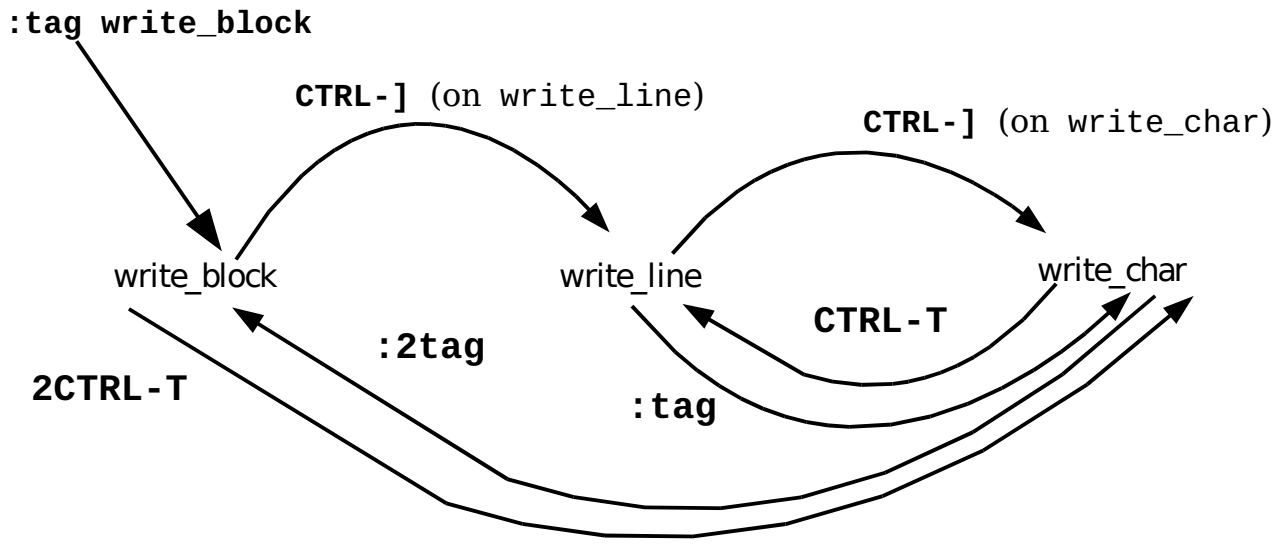


Figure 15: Tag navigation.

Hack 126: You Need to Find the Location of Procedure for Which You Only Know Part of the Name

The Problem: You "sort of " know the name of the procedure you want to find? This is a common problem for Microsoft Windows programmers because of the extremely inconsistent naming convention of the procedures in the Windows API. UNIX programmers fare no better. Only the inconsistency in naming conversions is consistent; the only problem is that UNIX likes to leave letters out of system call names (for example, `creat`).

The Hack: Use the **`:tag`** command to search for procedures using a regular expression.

If a procedure name begins with **`/`**, the **`:tag`** command assumes that the name is a regular expression. If you want to find a procedure named "something write something," for example, you can use the following command:

```
:tag /write
```

This finds all the procedures with the word write in their names and positions the cursor on the first one. If you want to find all procedures that begin with read, you need to use the following command:

```
:tag /^read
```

If you are not sure whether the procedure is `DoFile`, `do_file`, or `Do_File`, you can use this command:

```
:tag /DoFile\|do_file\|Do_File
```

or

```
:tag /[Dd]o_\=[Ff]ile
```

These commands can return multiple matches. You can get a list of the tags with the following command:

```
:tselect name
```

Figure 16 shows the results of a typical **`:tselect`** command.


```
~
    # pri kind tag       file
> 1 F C f write_char     write_char.c
                void write_char(char ch)
  2 F f write_block     write_block.c
                void write_block(char line_set[])
  3 F f write_line      write_line.c
                void write_line(char line[])
  4 F f write_raw       write_raw.c
                void write_raw(char ch)
Enter nr of choice (<CR> to abort):
```

Figure 16: ***:tselect*** *command.*

The first column is the number of the tag. The second column is the Priority column. This contains a combination of three letters.

F Full match (if missing, a case-ignored match)

S Static tag (if missing, a global tag)

F Tag in the current file

The last line of the **:tselect** command gives you a prompt that enables you to enter the number of the tag you want. Or you can just press Enter (**<CR>** in *Vim* terminology) to leave things alone.

The **g]** command does a **:tselect** on the identifier under the cursor. The **:tjump** command works just like the **:tselect** command, except if the selection results in only one item, it is automatically selected. The **gCTRL-]** command does a **:tjump** on the word under the cursor. A number of other related commands relate to this tag selection set, including the following:

:*count* tnext	Go to the next tag
:*count* tprevious	Go to the previous tag
:*count* tNext	Go to the next tag
:*count* trewind	Go to the first tag
:*count* tlast	Go to the last tag

Figure 17 shows how to use these commands to navigate between matching tags of a **:tag** or **:tselect** command.

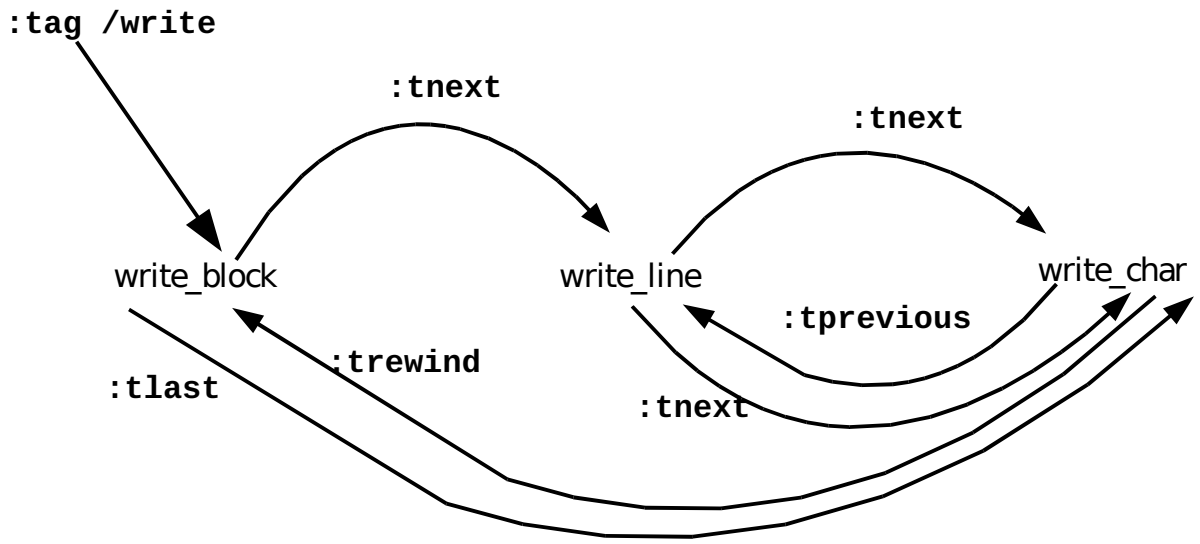


Figure 17: Tag navigation.

Hack 127: Use :vimgrep to Search for Variables or Functions

The Problem: You want to discover every place that a variable is used.

The Hack: Use *Vim's* internal **`:vimgrep`** command to do the searching for you.

The **`:vimgrep`** command acts much like **`:make`**. It runs the external program *grep* and captures the output. To find all occurrences of the variable ground_point, for example, you use this command:

```
:vimgrep /\<ground_point\>/ **/*.cpp
```

The `\<...\>` tells *Vim* to look for full words only. `ground_point` is the string you are looking for. Finally, there is the list of files to search through is all directories (**) and all C++ files (`*.cpp`). Figure 18 shows the results.


```
       ++i;
       return (0);
}
}

main.cpp:5: int i=3;
main.cpp:7: ++i;
sub.cpp:1:int sub(int i)
sub.cpp:3: return (i * j)
(1 of 4): : int i=3;
Press RETURN or enter command to continue
```

Figure 18: ***:vimgrep*** *output.*

Note: The *grep* program knows nothing about C++ syntax, so it will find `ground_point` even it occurs inside a string or comment.

You can use the **:cnext**, **:cprevious**, and **:cc** commands to page through the list of matches. Also **:crewind** goes to the first match and **:clast** to the last. Finally, the following command goes to the first match in the next file:

```
:cnfile
```

Hack 128: Viewing the Logic of Large Functions

The Problem: You've got legacy code that looks like:

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
}
```

How do you figure out the “logic” of such a function?

The Hack: Use the *Vim* fold feature.

Start by putting the cursor on the first line of the confusing code.

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
```



```
}
```

Press **V** to start visual line mode. The line will be highlighted.

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
}
```

Go to the end of the long code. You can do several ways:

1. Move the cursor using the normal cursor movement commands such as **j** (down) or **/** (search).

2. Go *up* to the curly bracket and press **%** (find matching bracket), then move up a line.

3. Enter the operator pending command **iB** to select the text inside the **{}**.

When you finish moving the cursor the lines inside the **{}** will be highlighted.

```
if (flag) {
    start_of_long_code();
    // 1,834 more lines of code
    end_of_long_code();
} else {
    return (error_code);
}
```

Now enter the command zf to "fold" the highlighted text. The 1,836 lines of code are now replaced by a single line telling you that a fold has been placed here.

```
if (flag) {
+--- 1836 lines: start_of_long_code(); -----------------+
} else {
    return (error_code);
}
```

The editor nows gives you an idea what the logic of the function looks like.

If you want to see the text again, position the cursor on the fold line and type **zo**.

There is a lot going on here, so you may wish to browse the help text. To get help on the commands documented execute the following commands:

Help Command	***Topic***
`:help V`	Visual Mode (the highlighting)
`:help %`	Match {} (and other things)
`:help v_iB`	Select inner braces (text between {})
`zf`	Create fold
`zo`	Remove (open) fold

Hack 129: View Logfiles with Vim

The Problem: You've got a log file, but it's 18,373 lines too long. How do you cut it down to size.

The Hack: Use your text editor. It's an ideal tool for text manipulation and browsing.

Let's take a look at some of the things you can do with the *Vim* editor. We'll assume you're already familiar with the simple commands like search and cut.

But there are other commands we can use. For example, let's suppose we are only interested in lines containing the string “Frame Header”. We can tell *Vim* to delete all lines except the ones containing “Frame Header” with the following commands:

1. Go to the first line in the code (**`1G`**)

2. Start a operation to pipe the text through a filter command. The portion of the text to be filtered that runs from the current cursor location to the end of the file (**`!G`**). <Exclamation Point><Letter G>

3. Enter the filter command. In this case we are using *grep* to search for lines so our filter command is: **`grep 'Frame Header'<enter>`**.

As one string this command is:

```
1G!Ggrep 'Frame Header'<enter>
```

The result is a log file with just the information we are interested in, and if we are lucky just the right amount of information to find the bug.

There are lots of ways of manipulating text in *Vim*. Far too many to list here.

The combination of extensive logs and a good text editor makes it easy to locate and view the information we want. This non-standard use of a text editor is one way a good hacker can think “out of the box” and make full use of the tools available to him.

Chapter 13: Clever but Useless

Not all hacks have a practical use. Some hackers invent very clever ways of doing things just to show off some very clever code so they can have bragging rights. Not that this is an entirely useless endeavor. Figuring how to say exchange the value of two variables without using a temporary requires you to know a lot about some of the more obscure aspects of the computer and of programming. And experimenting and learning these skills is one of the true attributes of a good hacker.

Hack 130: Flipping a Variable Between 1 and 2

The Problem: A variable `flag` can have the value `1` or `2`. If it is `1` change it to `2` and if it is `2` change it to `1`.

Now the most straight forward way of doing this is:

```
if (flag == 1) {
    flag = 2;
} else {
    flag = 1;
}
```

The paranoid hacker would use a **switch** and check for out of range values:

```
// This is the best and safest solution if we don't
// care about speed
switch (flag) {
    case 1:
        flag = 2;
        break;
    case 2:
        flag = 1;
        break;
    default:
        assert("flag is not 1 or 2" == 0);
        abort(); // Make sure we stop
}
```

The previous bit of code is the preferred solution if we don't care about speed and we wish to be both clear and paranoid.

But in this case our goal is to demonstrate how clever we are and produce a very clever hack.

The Hack: The variable can be toggled between the two values with one simple subtraction statement:

```
flag = 3 – flag;
```

You can now take a few seconds to verify the correctness of this statement.

This code functions however it is incomplete. If you must include clever code in your program like this, please please include comments telling the people that come after you what you've done.

```
// This clever piece of code flips the flag variable
// between the values of 1 and 2.
flag = 3 – flag;
```

This is almost the perfect hack. It is clever, it is small, it is extremely fast, and it is clear (after the addition of the comment). Now if we could just find some program we could use it in.

Hack 131: Swapping Two Numbers Without a Temporary

The Problem: How do you swap two variables without using a temporary.

The Hack: Use the following code:

```
void swap(int& a, int& b) {
    a ^= b;
    b ^= a;
    a ^= b;
}
```

This is the classic CS201 answer. But as hackers we are never content to leave things alone. Because I'm a hacker I wanted to see how things operate in the real world.

And I discovered another way of swapping variables without using a temporary:

```
static void inline swap(int& a, int& b) {
    int tmp = a;
    a = b;
    b = a;
}
```

Now you might wonder how I can claim that this doesn't use a temporary when I declare a temporary variable on the second line of the function. The trick is that I declared this function **inline** and turned on optimization. This left the compiler free to figure out the best way of swapping the two variables.

Let's take a look at what the optimizer did for us. The first test code looks like:

```
int main()
{
    int a = 1;
    int b = 2;

    print(a,b);
    swap(a,b);
    print(a,b);
    return(0);
}
```

Running this through the compiler and taking a look at the assembly code generated we can see that the compiler rewrote our main function:

```
    print(a,b);
    print(b,a);
```

The `swap` call was eliminated completely. So not only did the compiler eliminate the temporary, it also eliminated all the code as well. You can't get more optimized that that.

If we change our test to force the actual swapping of the variables we get the following code:

```
    mov     reg1, a
    mov     reg2, b
    mov     a, reg2
    mov     b, reg1
```

Now registers don't count as temporary variables (at least if I'm doing the counting) so again we have a case where we are swapping the variables without a temporary.

This question lets us illustrate several attributes of a true hacker. The first is a willingness to think outside the box. Anyone can look up the standard answer on the Internet. But only a good hacker will ask the question “Hey, what will happen if ....” and then perform a bunch of experiments answering that question. If the hacker is really good you find yourself with an answer that is surprising and much better than the “standard” answer.

Hack 132: Reversing the Words In a String Without a Temporary

The Problem: How do you reverse the word in a string without using a temporary string.

The Solution: Reverse the string, then go through the string reversing the words.

For example start with the string:

```
Now is the time
```

Reverse the entire string:

```
emit eht si woN
```

Then reverse the individual words:

```
time eht si woN
time the is woN
time the is Now
```

Turns out that moving a word from one end of a string to another is a difficult operation. Reversing a string is simple. So instead of moving words to their proper place, we reverse them in, then reverse the word to make it come out the right way round.

The code is below:

```
#include <iostream>
#include <ctype.h>

// Given two character pointers, reverse
// the string between them
static void rev_chars(
        char* ptr1,
        char* ptr2
)
{
    char tmp;

    while (ptr1 < ptr2) {
        tmp = *ptr1;
        *ptr1 = *ptr2;
        *ptr2 = tmp;
        ptr1++;
        ptr2--;
```



```
    }
}
// Reverse the words in a string
// (Without a temporary)
void reverse(char* const str)
{
    // Reverse the entire string
    rev_chars(str, str + strlen(str)-1);

    // Pointer to the first character of a word
    // to reverse
    char* first_ptr = str;

    // Keep looping until we run out of string
    while (*first_ptr != '\0') {
        // Move up to the first letter of the word
        while (! isalpha(*first_ptr)) {
            if (*first_ptr == '\0')
                return;
            first_ptr++;
        }
        char* last_ptr; // The last letter of the word + 1

        // Find last letter
        for (last_ptr = first_ptr+1;
                     isalpha(*last_ptr); ++last_ptr)
            continue;

        // Reverse word
        rev_chars(first_ptr, last_ptr-1);
        // Move to next
        first_ptr = last_ptr +1;
    }
}
```

Good hackers will look beyond the immediate problem and ask the question “Why are you doing such a thing?” It may be that the words are being consumed by some sort of command parser which needs them in reverse. If that's true, then there's no reason to reverse the words, then break them apart in the command parse. Just modify the command parser to break a string apart into words in reverse order.

Great hackers not only are able to solve the problem before them, but look beyond it and provide solutions that affect the bigger picture.

Government – The Anti-Hacker

A group of hackers were working for a government contractor translating a bunch of code from one version of JOVAL to another when they came across this very badly written function.

The figured that they could rewrite it to be both cleaner and more efficient. In order to make sure that their new design worked with the existing code, they searched the code to find every place where the function was called.

They found none. Nothing. The function was never called.

So they went to the boss and said, “This function is never used, we can get rid of it.”

The boss told them that he knew the function was never used. It hadn't been called for at least three releases. But because this was a government program, the cost of doing the paperwork far exceeded the cost of paying someone to update it.

Hack 133: Implementing a Double Linked List with a Single Pointer

The Problem: How can you implement a doubly linked list using only a single pointer for each link?

(This question was originally asked when memory cost $$$ instead of ¢ so at the time it made sense.)

The Hack: Exclusive or (XOR) the forward and backward links and store that value in the pointer:

```
node.link = next_ptr ^ prev_ptr;
```

When going through the list in a forward direction, you can reconstruct the pointer to the next node using the pointer to the previous one.

```
cur_ptr = first_ptr;
prev_ptr = NULL;

while (cur_node != NULL) {
    // Do something with the current node
```



```
    next_ptr = cur_ptr->link ^ prev_ptr;
    prev_ptr = cur_ptr;
    cur_ptr = next_ptr;
}
```

A similar system can be used to go through the list in reverse.

There are a few casts and a lot of details omitted from this code. Also I'm going to let the reader figure out to insert and remove noted.

However these days, memory is cheap, programmers are expensive, and it's far more cost effective to implement a double linked list as a double linked list, so this is just an academic exercise.

Hack 134: Accessing Shared Memory Without a Lock

The Problem: How do you synchronize two processes which share memory without using a test and set or lock instruction.

The Hack: There are several algorithms to do this. One of them is listed below.

This algorithm uses the following code to enter the critical section and access the shared resource:

1. Set a **`flag`** indicating that this process is now attempting to enter the critical section.

2. Set the **`turn`** variable to indicate that this process wants in. The **`turn`** variable will hold the ID of the last process to attempt entry.

3. If the other process wants the critical section and we were the last one to access the **`turn`** variable, wait. (A CPU consuming spin wait is used in this example.)

4. Do the critical stuff.

5. Set the **`flag`** to indicate that we are no longer in the critical section.

```
// Index for flag and turn variables
enum process {PROCESS_1 = 0, PROCESS_2 = 1};

// If true, I'm trying to go critical
volatile bool flag[2] = {false, false};

// Who's turn is it now
```



```
volatile int turn = PROCESS_1;

void process_1()
{
    flag[PROCESS_1] = true;
    turn = PROCESS_1;
    while (flag[PROCESS_2] && (turn == PROCESS_1))
         continue;     // do nothing

    do_critical();
    flag[PROCESS_1] = false;
}

void process_2()
{
    flag[PROCESS_2] = true;
    turn = PROCESS_2;
    while (flag[PROCESS_1] && (turn == PROCESS_2))
        continue; // do nothing

    do_critical();
    flag[PROCESS_2] = false;
}
```

As hackers we tend to think beyond the boundaries of the problem. In this example we careful to use the **volatile** keyword to make sure that the memory was actually changed and that the optimizer didn't play games with our code.

But hardware can play tricks on us too. Let's suppose these processes are running on different CPUs. If the processors have caches, the changes to **`flag`** and **`turn`** could be made in a local cache only and not copied to the shared memory. If this is the case we need to add some cache flush instructions to this system.

There are other questions to ask as well, such as why we don't have a some sort of hardware locking mechanism, or at least a test and set instruction. There are a lot of unanswered questions to think about.

But this problem is a good exercise when it comes to figuring out all the funny things that can happen in multiple process programming. You have to be aware that any process can run at any time and any shared variable can change at any time to be really effective when it comes to this type of programming.

Hack 135: Answering the Object Oriented Challenge

The Problem: Some people are fanatical about their programming methodology and refuse to acknowledge that any other system may be better.

A lot of people think that object oriented programming is the only way to go and if everybody used object oriented design, all the world's programming problems would be solved. As a long time hacker, I've lived through a number of fads, including expert programming, artificial intelligence, extreme programming, structured design, and many other. All of them promised to end the world's programming problems. None of them ever did.

One object oriented proponent once issued a challenge. “There is not one program that can not be made better through the use of objects,” he said. The challenge, find such a program.

The Hack: The *true* command.

In order to be fair, I choose as an example a real program that does real useful work. It's actually part of the Linux and UNIX standard command set. The command is *true.*

It's job is to return a status of *true* to the operating system for use in scripting. The following is a small example of shell scripting which uses this command:

```
if [ true ]; then
    echo “It is true”
fi
```

Now here is the C++ which I wrote without using object oriented design:

```
int main()
{
    return (0);
}
```

I'm looking forward to seeing how the object oriented guru can make this program better through the user of object.

Now some people might say I'm stretching the rules a bit. I am, but that's what hackers are supposed to do. We test the limits of the world we live in and sometimes we wind up making new limits.

We also don't lock ourselves into one technology just because someone says it's good. It has to actually be good for us to use it. As hackers the only thing we fanatical about is good code and will use anything to create it.

There are some other things to note about this program. First the C version of this code is exactly the same as the C++ one.

On UNIX this was originally implemented as a shell script containing 0 lines.

On my Solaris box, it is a shell script which has in it nothing but five copyright notices, a warning telling me that the contents are unpublished prosperity code, and a version number: 1.6. No commands.

The version number is most interesting, since the script contains no code. What did they get wrong in versions 1.0 through 1.5 that got fixed in 1.6?

The version of the command that ships with Linux contains options. One prints out the version number of the command, the other prints out a help text that tells you that the only option is the one to print out the version number. Incidentally the version number on my system is 5.2.1. Again one has to wonder what was wrong with the previous versions?

So the *true* command serves two purposes. One it shows how absolute statements such as “I can make it better with objects” aren't always true. And it also shows how managers, copyright lawyers, and other forces can add needless complexity to a program that the closest thing to a nothing program that I know of.

Appendix A: Hacker Quotes

The speed of a non-working program is irrelevant. -- Unknown.

Profanity is the one language all programmers know best. -- Unknown.

Don't believe the requirements, management, or the users. Program as if they they don't know what they really want. Usually they don't. -- Steve Oualline

The purpose of a schedule is to tell you how late you really are. -- Unknown

Oualline's law of documentation: 90% of the time the documentation will be lost. Out of the remaining 10%, 9% of the time it will be the wrong version. The one time you have the correct documentation and the correct version of the documentation, it will be written in Chinese.

O'Shea's law: Murphy was an optimist.

Its not possible for a program to meet requirements unless the requirements have actually been defined. -- Steve Oualline

Its not possible to meet schedule unless you have one. -- Steve Oualline

Why is there never enough time to do it right, but always enough time to do it over. -- Unknown

The first 90% of your program will take 90% of your allocated time to create. So will the last 10%. -- Steve Oualline

"To be or not to be, --that is the question." Shakespeare on boolean algebra. Hamlet, Act III, Scene 1.

"Bloody Instructions, which being taught, return to plague the inventor," Shakespeare on maintenance programming. Macbeth Act 1, Scene 6.

Grace Hopper

Inventor of the first compiler (COBOL) and well known for finding a moth beaten to death by the contacts of a relay based computer. (This incident has given rise to the legend (incorrect) that this was the first computer bug)

The wonderful thing about standards is that there are so many of them to choose from.

"The most damaging phrase in the language is: We've always done it this way."

If it's a good idea, go ahead and do it. It's much easier to apologize than it is to get permission.

Linus Torvalds

Author of Linux.

Any program is only as good as it is useful.

I'm generally a very pragmatic person: that which works, works.

In many cases, the user interface to a program is the most important part for a commercial company: whether the programs works correctly or not seems to be secondary.

Linux tends to have fewer rules than other developments, and anybody can chip in doing whatever they want.

Microsoft isn't evil, they just make really crappy operating systems.

The cyberspace earnings I get from Linux come in the format of having a Network of people that know me and trust me, and that I can depend on in return.

The memory management on the PowerPC can be used to frighten small children.

See, you not only have to be a good coder to create a system like Linux, you have to be a sneaky bastard too.

If you need more than 3 levels of indentation, you're screwed anyway, and should fix your program.

You know you're brilliant, but maybe you'd like to understand what you did 2 weeks from now.

Talk is cheap. Show me the code.

Really, I'm not out to destroy Microsoft. That will just be a completely unintentional side effect.

The fact that ACPI was designed by a group of monkeys high on LSD, and is some of the worst designs in the industry obviously makes running it at any *point pretty damn ugly.*

Given enough eyeballs, all bugs are shallow.

I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.

Appendix B: You Know You're a Hacker If...

You know you're a hacker if...

your house has more computers in it than TVs.

you know at least seven people by their e-mail address, but have forgotten their names.

you've watched a movie of two people making love on a computer desk and tried to figure out what kind of computer was in the background.

you have memorized the number of at least one pizza place which will deliver to the computer lab after midnight,

you've lived on ramen noodles for more than a week.

you've disassembled a program to see why it ticked

you've gone up to a complete stranger and asked them what sort of laptop they are using,

everyone in your family comes to you with their computer problems

you taken off the covers of all your computers at least once.

you named your daughter after Grace Hopper

Grace Oualline (Age 3)

you've ever use the term TCP/IP during a date.

your wife has stood naked next to the computer and asked you "Do you want to work on the computer or make love?" and you had to think about the answer.

you thought about the answer then decided to fix just one more bug.

you've edited a program on a laptop you brought to Disneyland.

played computer games for less than an hour before trying to disassemble the thing to figure out how it worked.

once you figured out how the game worked, decided to improve it.

wired a household appliance to a computer.

ever said “I now know the difference between & and &&” to someone who understood exactly what you meant. (& is pronounced “and”.)

Appendix C: Hacking Sins

Using the letters O, l, I as variable names

In case you didn't notice we are talking about the upper case letter “O”, the upper case letter “I” and the lower case letter (l) (Lower case L).

The fact that I have to include the previous sentence in this book should tell you how easily it is to mistake these variable for something else. For example, can you tell the difference between the letter 0 and the number O. If you can, then you know I got them backwards in the previous sentence.

Not Sharing Your Work

A big part of being a hacker is helping other people become hackers, or at least help them make more effective use of their computers. The GPL is all abort sharing. You can use the work of others as long as you share your work with them.

No Comments

I've heard that the next great programming language is going to be so easy to write in that you won't need comments. Unfortunately I heard that 30 years ago and the language in question was FORTRAN II. They also said it about COBOL. They were wrong.

English is still one of the best ways of communicating with people. Not using it to explain what you do means that you've created a cryptic mess which can't be shared by other people.

Inc0nsisTencY

When you program consistently you create patterns which allow other people to easily understand and augment your code. Inconsistency creates confusion and disrupts the work process.

(Inconsistency is book writing means that you have to explain to copy editor why your strange formatting is really just an attempt to humor. They keep missing the joke and trying to fix it.)

Duplicating Code (Programming by Cut and Paste)

There is a type of programmer out there who when faced with a problem, will simply use his text editor to copy some code that does something similar, make a few changes, and release his solution. 95% of his "new" code is copied and 5% is original work (if that much). This type of programmer is called a hack. (In spite of the similarities in the names, "hack" and "hacker" are complete opposites.)

A hacker modifies the existing function to make it more general and more useful to a large audience.

Cut and paste programming is one way of creating large programs which are impossible to maintain. Hackers like small, well crafted, well designed code, instead of quickly constructed large messes.

Appendix D: Open Source Tools For Hackers

ctags – Function Indexing System

The original UNIX *ctags* was rather limited, so Exuberant Ctags was created. It has many extended features beyond the basic *ctags* functionality.

Written by Darren Hiebert it can be found at http://ctags.sourceforge.net/.

doxygen

A system for embedding documentation inside a C or C++ program (as well as some other languages.) When used on a whole system this is a very effective tool for generating code documentation.

Available from http://www.doxygen.org

FlawFinder

Analyzes code for potential security problems.

Available from http://www.dwheeler.com/flawfinder/

gcc – The GNU C and C++ compiler suite

This is a high quality with lots of extensions which make programming easier and more reliable. It was written by hackers for hackers and it shows.

Available at http://www.gnu.org.

lxr

Linux cross reference. Actually this tool will cross reference any program and produce a set of hyper-linked files which make it easy to navigate through large programs.

The program can obtained at: http://lxr.linux.no/.

Perl (for perldoc and related tools) – Documentation System

Perl contains tools for manipulating documents in POD (Plain Old Documentation) format. This format is the standard way of documenting Perl code, but it works for C and C++ as well.

You can get perl from http://www.perl.org.

valgrind (memory checking tools)

A tool which checks for memory corruption and as well as the use of uninitialized memory.

You can download the program from: http://www.valgrind.org.

Vim (Vi Improved)

A text editor. Having said that, it does its job very very well and contains lots of features designed to make programming easier. Unlike integrated development environments (IDEs) which try do multiple jobs, *Vim* edits text and that's it. True it does have interfaces to compilers and other tools, but its main job is text editing.

The learning curve is a little steep for *Vim* but once you learn how to use it you can edit files quicker than any other editor.[25]

Vim can be downloaded from http://www.vim.org.

25 People who love the EMACS editor and others may dispute this claim.

Appendix E: Safe Design Patterns

1. Eliminate side effect. Put ++ and -- on lines by themselves.

```
++i;      // Good
--i;      // Good

i++;      // Good but inefficient (See ###)

a = i++;  // Bad
a[i] = i++;    // VERY bad
```

2. Don't put assignment statements inside other statements

```
// Bad and probably wrong
if (x = 5) ...

// Good
x = 5;
if (x != 0) ...
```

3. Defining Constants

Use **const** if possible:

```
const int WIDTH = 50;
```

If you must use **#define** put the value in ():

```
#define AREA (50 * 10)
```

4. Use inline functions instead of parameterized macros

```
inline int square(int i) { return (i*i); }
```

If you must use parameterized macros, put parentheses () around each argument.

```
#define SQUARE(i) ((i) * (i))
```


5. Use {} to eliminate ambiguous code

```
// Very bad
if (a) if (b) do_alpha() else do_beta();

// Good and indented too
if (a) {
    if (b) {
        do_alpha();
    } else {
        do_beta();
    }
}
```

Some people (especially Perl programmers) say you should always include the curly braces.

6. Be obvious about everything you do, even nothing

```
for (i = 0; buffer[i] != 'x'; ++i)
    continue;

switch (state) {
    case 1:
        do_stage_1();
        // Fall throught
    case 2:
        do_stage_2();
        break;
    default:
        // Should never happen
        assert("Impossible state" == 0);
        abort();
}
```

7. Always check for the default case in a switch

See above.

8. Precedence Rules

1 Multiply (*) and divide (/) come before addition (+) and subtraction (-).

2 Put parentheses () around everything else.

9. Header Files

Always include your own header file.

```
/* File: report.c */
...
#include “report.h”
```

Protect against double inclusion:

```
/* File report.h */
#ifndef __REPORT_H__
#define __REPORT_H__
// ....
#endif /* __REPORT_H__ */
```

10. Check User Input

When checking user input always check for what's *permitted,* never check for what's *excluded*. That way if you make a mistake you may accidentally prevent the inclusion of some good data, but you don't let bad data through.

11. Array Access

```
assert((index >= 0) &&
       (index < (sizeof(array) / sizeof(array[0]))));
value = array[index]; // or
array[index] = value;
```

12. Copying A String

```
strncpy(dest, source, sizeof(dest));
```

13. String Concatenation

```
strncat(dest, source, sizeof(dest) – strlen(dest));
dest[sizeof(dest)-1] = '\0';
```

14. Copying Memory

```
memcpy(dest, source, sizeof(dest));
```

15. Zeroing memory

```
memset(dest, '\0', sizeof(dest));
```


16. Finding the Number of Elements in an Array

```
n_elements = sizeof(array) / sizeof(array[0]);
```

17. Using gets

Don't use **gets**. There is no way to make it safe. Use **fgets** instead.

18. Using fgets

```
fgets(string, sizeof(string), in_file);
```

19. When creating opaque types, make them checkable by the compiler

```
struct font_handle {
    int handle;
};

struct window_handle {
     int handle;
}

struct memory_handle {
     int handle;
}
```

20. Zero pointers after delete/free to avoid reuse

```
delete ptr;
ptr = NULL;

free(b_ptr);
b_ptr = NULL;
```

21. Always check for self assignment

```
class a_class {
    public:
        a_class& operator = (a_class& other) {
            if (&other == this) return (other);
        }
```

22. Use snprintf to create string

```
snprintf(buffer, sizeof(buffer), “file.%d”, sequence);
```


Appendix F: Creative Commons License

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

1. Definitions

a. **"Collective Work"** means a work, such as a periodical issue, anthology or encyclopedia, in which the Work in its entirety in unmodified form, along with one or more other contributions, constituting separate and independent works in themselves, are assembled into a collective whole. A work that constitutes a Collective Work will not be considered a Derivative Work (as defined below) for the purposes of this License.

b. **"Derivative Work"** means a work based upon the Work or upon the Work and other pre-existing works, such as a translation, musical arrangement, dramatization, fictionalization, motion picture version, sound recording, art reproduction, abridgment, condensation, or any other form in which the Work may be recast, transformed, or adapted, except that a work that constitutes a Collective Work will not be considered a Derivative Work for the purpose of this License. For the avoidance of doubt, where the Work is a musical composition or sound recording, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered a Derivative Work for the purpose of this License.

c. **"Licensor"** means the individual, individuals, entity or entities that offers the Work under the terms of this License.

d. **"Original Author"** means the individual, individuals, entity or entities who created the Work.

e. **"Work"** means the copyrightable work of authorship offered under the terms of this License.

f. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

2. Fair Use Rights. Nothing in this license is intended to reduce, limit, or restrict any rights arising from fair use, first sale or other limitations on the exclusive rights of the copyright owner under copyright law or other applicable laws.

3. License Grant. Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to reproduce the Work, to incorporate the Work into one or more Collective Works, and to reproduce the Work as incorporated in the Collective Works;

b. to create and reproduce Derivative Works provided that any such Derivative Work, including any translation in any medium, takes reasonable steps to clearly label, demarcate or otherwise identify that changes were made to the original Work. For example, a translation could be marked "The original work was translated from English to Spanish," or a modification could indicate "The original work has been modified.";;

c. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission the Work including as incorporated in Collective Works;

d. to distribute copies or phonorecords of, display publicly, perform publicly, and perform publicly by means of a digital audio transmission Derivative Works.

e. For the avoidance of doubt, where the Work is a musical composition:

i. **Performance Royalties Under Blanket Licenses**. Licensor waives the exclusive right to collect, whether individually or, in the event that Licensor is a member of a performance rights society (e.g. ASCAP, BMI, SESAC), via that society, royalties for the public performance or public digital performance (e.g. webcast) of the Work.

ii. **Mechanical Rights and Statutory Royalties**. Licensor waives the exclusive right to collect, whether individually or via a music rights agency or designated agent (e.g. Harry Fox Agency), royalties for any phonorecord You create from the Work ("cover version") and distribute, subject to the compulsory license created by 17 USC Section 115 of the US Copyright Act (or the equivalent in other jurisdictions).

f. **Webcasting Rights and Statutory Royalties**. For the avoidance of doubt, where the Work is a sound recording, Licensor waives the exclusive right to collect, whether individually or via a performance-rights society (e.g. SoundExchange), royalties for the public digital performance (e.g. webcast) of the Work, subject to the compulsory license created by 17 USC Section 114 of the US Copyright Act (or the equivalent in other jurisdictions).

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats. All rights not expressly granted by Licensor are hereby reserved.

4. Restrictions. The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may distribute, publicly display, publicly perform, or publicly digitally perform the Work only under the terms of this License, and You must include a copy of, or the Uniform Resource Identifier for, this License with every copy or phonorecord of the Work You distribute, publicly display, publicly perform, or publicly digitally perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of a recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties. When You distribute, publicly display, publicly perform, or publicly digitally perform the Work, You may not impose any technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collective Work, but this does not require the Collective Work apart from the Work itself to be made subject to the terms of this License. If You create a Collective Work, upon notice from any Licensor You must, to the extent practicable, remove from the Collective Work any credit as required by Section 4(b), as requested. If You create a Derivative Work, upon notice from any Licensor You must, to the extent practicable, remove from the Derivative Work any credit as required by Section 4(b), as requested.

b. If You distribute, publicly display, publicly perform, or publicly digitally perform the Work (as defined in Section 1 above) or any Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above), You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or (ii) if the Original Author and/or Licensor designate another party or parties (e.g. a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; the title of the Work if supplied; to the extent reasonably practicable, the Uniform Resource Identifier, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work; and, consistent with Section 3(b) in the case of a Derivative Work, a credit identifying the use of the Work in the Derivative Work (e.g., "French translation of the Work by Original Author," or "Screenplay based on original Work by Original Author"). The credit required by this Section 4(b) may be implemented in any reasonable manner; provided, however, that in the case of a Derivative Work or Collective Work, at a minimum such credit will appear, if a credit for all contributing authors of the Derivative Work or Collective Work appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED TO BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND ONLY TO THE EXTENT OF ANY RIGHTS HELD IN THE LICENSED WORK BY THE LICENSOR. THE LICENSOR MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MARKETABILITY, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability. EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Derivative Works (as defined in Section 1 above) or Collective Works (as defined in Section 1 above) from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You distribute or publicly digitally perform the Work (as defined in Section 1 above) or a Collective Work (as defined in Section 1 above), the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted to You under this License.

b. Each time You distribute or publicly digitally perform a Derivative Work, Licensor offers to the recipient a license to the original Work on the same terms and conditions as the license granted to You under this License.

c. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

d. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

e. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

Creative Commons Notice

Creative Commons is not a party to this License, and makes no warranty whatsoever in connection with the Work. Creative Commons will not be liable to You or any party on any legal theory for any damages whatsoever, including without limitation any general, special, incidental or consequential damages arising in connection to this license. Notwithstanding the foregoing two (2) sentences, if Creative Commons has expressly identified itself as the Licensor hereunder, it shall have all rights and obligations of Licensor.

Except for the limited purpose of indicating to the public that the Work is licensed under the CCPL, Creative Commons does not authorize the use by either party of the trademark "Creative Commons" or any related trademark or logo of Creative Commons without the prior written consent of Creative Commons. Any permitted use will be in compliance with Creative Commons' then-current trademark usage guidelines, as may be published on its website or otherwise made available upon request from time to time. For the avoidance of doubt,

