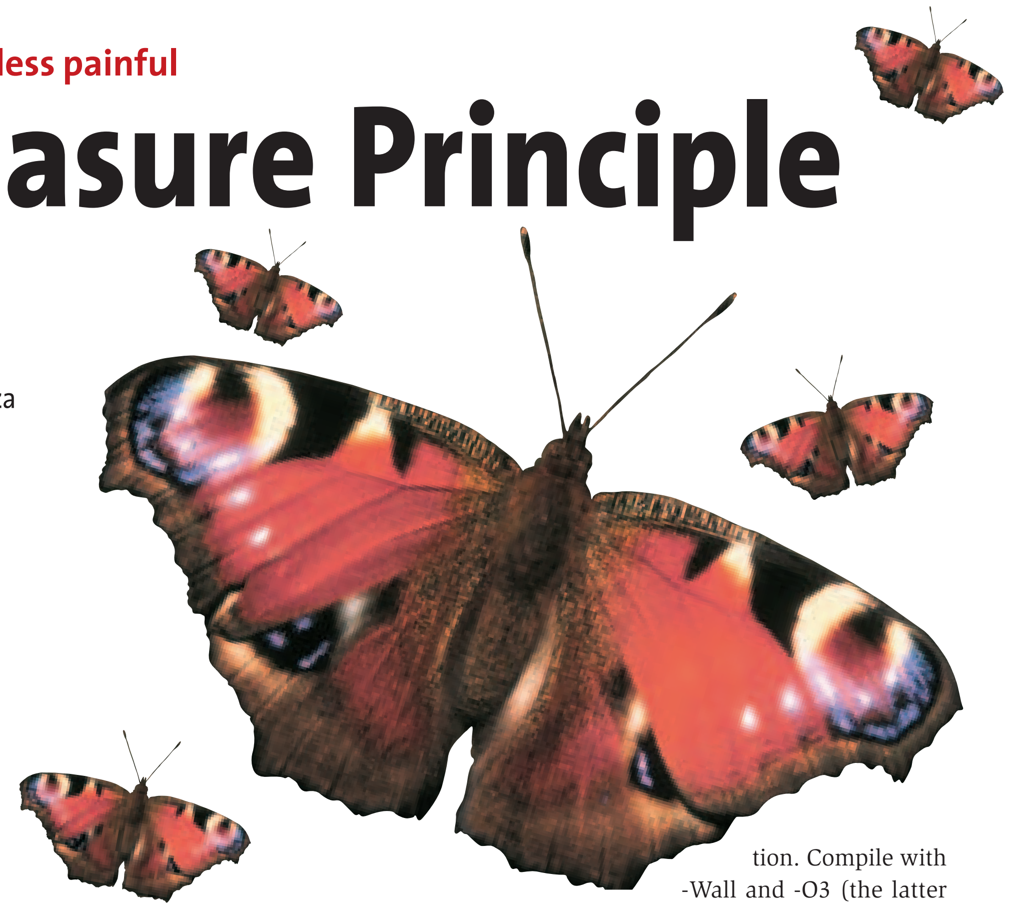
How to make debugging less painful

The Pleasure Principle



Debugging software is a necessary evil in development. Most programmers would forgo a year of pizza if they never had to fix a bug again. With the right tools, and a little patience, the task becomes much easier. This article covers some of these tools, and looks more in depth at the practises involved with one of the most prominent, gdb.

BY STEVEN GOODWIN

All bugs stem from one basic premise – something you thought was right, was in fact wrong. This could range from an assumption about the value of variables, to the validity of a memory address or pointer. The effect of these beliefs, however, vary enormously. It might produce a segmentation fault, hang in an endless loop, or report invalid results. But *all* bugs are important, and we should do our best to eradicate them all.

The conventional way of solving such problems is to output trace messages from within your program. This involves adding code like:

```
fprintf(stderr, "Entering ➘
CalcAverage now (num=%d, ➘
total=%d)",num,total);
...
fprintf(stderr, "Leaving ➘
CalcAverage now (return av=%d)"➘
, average);
```

We can then watch the stderr output stream to pinpoint which function goes wrong, and from there we can look further to find out *why*.

Unfortunately, while suitable for small programs, adding code like this quickly becomes tiresome, and therefore error prone. It's also a very static approach – you can see the error happening, but are unable to do anything about it. There are various tools we can use that do not require so much work, nor do they require much (if any) modification to the original source code.

Low Rider

Laziness is a good virtue for the programmer. As is procrastination. Before jumping into the debugging process immediately, it is always a good idea to take a step back and consider preventative measures.

Firstly, does it compile cleanly? That is, does the compiler produce any warnings, and if so in what way could they cause the program to malfunction. Compile with -Wall and -O3 (the latter does extra checks that non-optimised builds don't) and re-check the output.

Secondly, run other code review tools, such as SPLint (Linux Magazine, issue 27), to report on errors that gcc doesn't. This will highlight code that typically fails to work in all cases, or demonstrates gaps in the implementation that need to be catered for. Typical problem cases are:

- An unsigned number being used in an < 0 case (which can never be true)
- The equality symbol (==) being mistyped as an assignment (=)
- Type casting (explicit or otherwise) that might lose information

Thirdly, do any other tools report prob-

Table 1: General Commands

Shortcut	Command	Description
r	run [args]	Begin a program
c	continue	Continue running, from current breakpoint
n	next	Next instruction, stepping over function calls
s	step	Step to next line or function
p	print	Display a variable
pt	ptype	Gives the type of a variable. Also details structures
bt	backtrace	Lists a call stack of all functions called to reach here (also shown as where)
l	list	Lists part of the program
h	help	Specific help available with 'help delete', for example
q	quit	Leave GDB

lems with the program when run. For example, memory debuggers that check for malloc problems, and accesses to data outside its valid bounds. Most commonly this occurs with strings (as their bounds are rarely checked), but can happen with any array, pointer, or dynamically allocated memory.

Programs in this category include Valgrind, *ccmalloc*, and Electric Fence by Bruce Perens (of Debian fame). This list is not complete, and not all tools function in the same way. Valgrind, for example uses just-in-time (JIT) debugging and is based on code re-interpretation. *ccmalloc*, on the other hand, requires your compiled object code to be linked with the ccmalloc library, in order to log memory leaks.

If, after all this, your program still refuses to work we need to work through the code line-by-line to find out where (and why) it is going wrong. The most oft-used tool for this is gdb (or one of its many graphic counterparts). It allows you to interactively control the running of the program, stop it at various times, inspect the variables, and even change the flow of code whilst running. This allows you to confirm that the program *will* be correct, once the current error has been fixed.

Enter The Dragon

GDB stands for the GNU Debugger, and is one of the older parts of the development suite and was started by rms in 1988. Despite (or perhaps) because of, its history, gdb is still a command line application and comes complete with a wealth of powerful commands.

However, many people have created graphical components around it, to make the debugging environment a little friendlier. Most Linux distributions come with several of these (such as ddd, gvd and xxgdb), along with a couple of other variations on a theme.

The Linux Kernel, for example, has it own special version (kgdb) that allows remote kernel debugging across a serial line. There's also gdbserver for controlling gdb across a TCP/IP connection. For the purposes of this article however, we shall concentrate of the grandfather of all these tools, gdb.

In order to use GDB you must compile your program with special GNU debugging information into the program. This data is known as 'symbol information' and describes (amongst other things) where the functions and variables are stored in memory. This makes it possible for the debugger to give you (the programmer) detailed information about your program.

To add symbol information to your program, simply re-compile with the -g flag, thus:

```
$ gcc -g -Wall -O3 change.c ➥
-o change
```

Currently, the C, C++ and Modula-2 compilers support GDB debugging. We shall be using the simple C program given in Listing 1, as our test case. It reports the fewest coins required to give change for a given value.

A program that is compiled with the -g option can run be run as if it were any normal executable, and you would be hard pushed to notice any difference. However, the program now has had some 'magic dust' added to it that allows you to run it through the gdb debugger. You can then load the debugger and program in one go with the command:

```
$ gdb change
```

This will load gdb and automatically bring *change* into memory for debugging: but it won't start running it until you say so! (see also Box 1 „Nancy Boy") GDB works from its own com-

Listing 1: Simple Coin program

```
#include <stdio.h>
#include <stdlib.h>

void CalcChangeFor(int pence)
{
int i, val;
int coins[] = { 500,200,100,50,20,10,5,2,1,};

        if (pence == 0)
                return; /* amount is exact */
        if (pence < 0)
                return; /* invalid amount */

        for(i=0;i<8;i++) {    /* for each coin */
                val = coins[i];
                for(i=pence;i>0;i++)
                        if (val*i <= pence)
                                printf("%d x %dp\n", i, val);
                                CalcChangeFor(pence - i*val);
                                return;
                }
}

void usage(void)
{
        fprintf(stderr, "Usage: change <amount>\n");
}

int main(int argc, char **argv)
{
        if (argc > 1)
                CalcChangeFor(atoi(argv[1]));
        else
                usage();
        return 0;
}
```

mand prompt (which also features command history) and uses Linux-style commands (short, often cryptic, and usually abbreviated) to control the various features and options.

For example, 'run' will start the currently loaded program from the beginning, and can be shortened to 'r'. Like most gdb commands this may contain optional arguments. For instance:

```
(gdb) r 23
```

This will run the current program, passing the number 23 as its first argument. But don't run it yet because the example code is (naturally enough) full of bugs! So, let us work through the program inside the debugger to try and find them.

Stepping Stone

In order to examine the workings of the program, we need to find somewhere to start. Or rather, somewhere to stop! We need to find a line of code that we can start working from. In this example, we'll pick the first line of our program. Instead of having to refer to the listing in an external editor, we can use the internal 'list' command to show us the source for any location in our project (see Listing 2).

This will list the main function (plus a few lines before it) to the screen. This information is taken directly from the source code itself and so needs to be present on the machine.

If you press return (without specifying a command) gdb will re-interpret the last command given. In the case of list, it will continue to list the next few lines of the source code. If the previous command was 'execute the next statement', you could step through the entire program pressing only the return key. This is a feature you will appreciate over time!

From the listing we can determine that we should stop the program running at line 31. Once stopped, gdb offers us a command line with which to interrogate the programs variables and functions. We can then work through the function one line at a time in order to discover what went wrong.

In order to stop a program, we use what is known as a *breakpoint*. The program runs normally until it is about to execute the piece of code at the same address as the breakpoint (it is said to have *hit* the breakpoint) at which point it drops back into the debugger for us to look at variables, or continue stepping through the code.

```
(gdb) break change.c:31
Breakpoint 1 at 0x80485e0: file
 change.c, line 31.
```

Breakpoints are fundamental in interactive debugging, and accordingly have many options associated with them. They can be set-up on a specific line number (as we've done here), at the beginning of a function, at a specific address, or conditionally when a loop counter reaches 906 (for example).

There can be any number of breakpoints within the same program, and they need not be created before the program is run. They can be added or removed at any time from the gdb command prompt. We can also list the breakpoints currently in use with the command as seen in Listing 3.

Other oft-used breakpoint commands can be found in table 2.

Table 2: Breakpoint commands

b [filename:]line_number	Place breakpoint on line
b [filename:]function	Place breakpoint at start of function
b [filename:]line_number if expr	Place conditional breakpoint
watch *expression*	Add watchpoint on variable/expression (shortens to wa)
info break	Lists breakpoints and watchpoints
delete	Delete all breakpoints
delete [N]	Number breakpoint #N (as listed by info break)
disable [N]	Disable breakpoint (or all if omitted)

Keep on Moving

Once the program has stopped running, there are three essential commands to get it moving again: 'n', 's' and 'c'. 'n' executes the next instruction, stepping over any function call it comes across and returns to the gdb prompt. Think of the letter 'n' as a bridge that crosses over something, to distinguish it with the next command.

's' will step to the next instruction, whether it is inside a function or not. Again, it will return to the prompt afterwards, And finally, 'c' will continue running the program until the next breakpoint is reached, or the program has terminated. Since the standard libraries are not built with debugging information, it is not possible to step inside them using the 's' command.

So let us step into our CalcChangeFor function. It passes all the validation functions without a problem, and starts working on the loop to find the maximum number of 5 pound notes that can be given in change for 23 pence. Naturally, there shouldn't be any. However, once inside the loop, it seems reluctant to leave! Let us look at the variables to give us a clue how.

Changes

Interrogating variables is a similarly easy task. The symbol information in the exe-

Box 1: Nancy Boy

There are several ways to launch the gdb debugger. Including the program name as an argument is one way, using the 'file' command is another. It is also possible to attach it to an already running process, or by using a 'core dump'. A core dump details the memory during the last moment of the programs life, in a similar way to an aircrafts 'black box' recorder.

Listing 2: Part examination

```
(gdb) list change.c:main
25      {
26              fprintf(stderr, "Usage: change <amount>\n");
27      }
28
29      int main(int argc, char **argv)
30      {
31              if (argc > 1)
32                      CalcChangeFor(atoi(argv[1]));
33              else
34                      usage();
```

cutable (added by the -g option) contains full details of the variables, their types and program scope (if they are local or global). We can print out the value of any variable in scope (and only those in scope) with the ‘p’, or ‘print’, command. So when we have stepped into, and up to, line 17 we can type:

```
(gdb) print val
$1 = 500
(gdb) p i
$2 = 23
```

The ‘$‘ symbols on the left refer to the most recently displayed values, and will increment with each command.

If we want a more permanent record of these variables, we can ask gdb to output them after every command by using the *display* command (other samples of which can be found in table 3).

Table 3: Display commands

print [/F] [expr]	Print expression immediately. /F is an optional format, e.g. /x = hex, /d = dec, /u = unsigned /o = oct, /t = bin, /c = char, /f = float
display [/F] [expr]	Print expression before every command prompt
display	List values for all expressions
undisplay	Remove all expressions from display list
undisplay [N]	Remove specific expression from list (as given with display command)

```
(gdb) display val
1: val = 500
(gdb) disp i
2: i = 23
```

We can then continue stepping through the code (using ‘s’), to witness these values in an attempt to understand why the loop doesn’t terminate.

It shouldn’t take us long before we realise the error, since we can now see the *i* variable increment instead of decrement: there’s a + + instead of a – in the increment portion of the *for* loop. A simple mistake, but one made much easier to find with the variable being listed at each step. We fix our code, and reload.

The Unforgiven II

Because loading the gdb debugger is a time consuming task, there is a trick that allows us to start the program again, without quitting gdb itself. Firstly, we kill the current process:

```
(gdb) kill
Kill the program being ➥
debugged? (y or n) y
```

We then recompile our program, and use the ‘file’ command to load it back into memory, before running it in the normal way. In this way we can keep all our previous breakpoints, saving us set-up time that will become quite significant on larger projects. If you try to re-compile, and find that the file is ‘busy’ that means the debugger is still using it and you forget to kill the process first.

```
(gdb) file change
Load new symbol table from ➥
"change"? (y or n) y
Reading symbols from ➥
"change"...done.
(gdb) r
```

(Note that ‘r’ retains its previous argument, so we don’t need to re-type ‘r 23’)

This time if we step through our loop, we see that *i* is counting down correctly. So we remove all our breakpoints (see TABLE 1: Breakpoint Commands), and continue. After an unusually long time the debugger produces a segmentation fault and issues a command prompt (see Listing 4).

Hmmm. There must be another bug. To see where we are in the program we need to look at the call stack, or backtrace, to see what functions have been called, and with what parameters (see Listing 5).

This tells us that our algorithm is getting stuck in a recursive loop: it is calling the same function time and time again with the same parameter. Consequentially, the exit condition (pence = = 0) is never triggered. So how does this happen?

If we restart the debugger (we haven’t changed the file this time, so a simple ‘kill’ and ‘run’ will suffice), and put breakpoints on each recursive call to CalcChangeFor we can see that the value of ‘pence’ goes through unaltered each time because i*val = 0. Looking at the variables of *i* and *val*, we see that, although *val* represents the correct coinage (5 GBP) *i* is zero. Since that should only happen once the entire loop has completed, and the values of *i inside* the loop must always be greater than 0 (because the termination condition on line 16 reads, i > 0) we can deduce that we’re not actually inside the loop anymore.

A check of the code reveals that there are no braces around lines 18 to 20. Although the formatting might suggest it to us, the reader, the compiler does not see this, and so introduces this current bug.

Another restart, another bug! This time, we get no output, nor any usable call stack. So, what do we do?

We have a reasonable idea that the program is stuck in a loop, and since we only have two of them, it should be fairly easy to track down. We start by putting a breakpoint on the outer loop (the loop of each coin from line 14 to 22) and see if that loop iterates correctly. If it doesn’t return to the start then we know it’s the loop inside that. If it does, then it’s a problem with the outer loop.

So, we add our breakpoint at line 15 (see Box 3 “Numbers“ for why this breakpoint is called 2, and not 1), and restart gdb. The breakpoint fires! See Listing 6.

Now we are inside the function, the loop counter *i* is in scope. And because only in-scope variables can be printed or displayed we can now see the loop count, and work our away around the loop, observing its behaviour. So let us continue:

Listing 3: Breakpoints

```
(gdb) info break
Num Type           Disp Enb ➥
Address     What
1   breakpoint     keep y   ➥
0x080485e0 in main at change.c:31
```

Listing 4: Segmentation fault

```
Program received signal SIGSEGV, Segmentation fault.
0x080485a7 in CalcChangeFor (pence=23) at change.c:19
19                                        CalcChangeFor(pence - i*val);
```

```
(gdb) disp i
3: i = 0
(gdb) c
Continuing.

Breakpoint 2, CalcChangeFor ➮
(pence=23) at change.c:15
15          val = coins[i];
3: i = 1
```

The first iteration appears fine. Let's continue:

```
(gdb) c
Continuing.

Breakpoint 2, CalcChangeFor ➮
(pence=23) at change.c:15
15          val = coins[i];
3: i = 1
```

The second iteration did not appear to happen because *i* hasn't been incremented! Since we can see the i++ expression in line 14 we're sure it must be getting incremented correctly – it's just that something else is decrementing it *in*correctly. At this stage we bring on the big guns – watchpoints.

These are powerful 'breakpoints on steroids' features which will stop the code whenever a variable change – on whichever line it occurs – even if the line doesn't reference the variable explicitly by name! Instead, it looks at the memory address of the variable and alerts you when something is written into it.

```
(gdb) watch i
Hardware watchpoint 3: i
```

If we continue running, we'll see whenever *i* is changed:

```
(gdb) c
Continuing.

Hardware watchpoint 3: i
```

Box 2: Close To Me

The debugger should not just be consigned in the aftermath of coding. It should become a natural part of your development cycle. Because prevention is better than cure, step through all newly written code with a debugger. Think about every line. "Does it execute every line? Does it handle the error conditions? (Remember that keyboard I/O is also a stream that can return an EOF, not just files!) Do the loops terminate correctly?"

These are just some of the questions you should ask yourself and check with your debugger before running the program to see 'if it works yet'. Because if you can step through the code, and it follows the code paths you expect, it probably will work first time. And that is a great feeling.

```
Old value = 1
New value = 23
0x080484db in CalcChangeFor ➮
(pence=23) at change.c:16
16          for (i=pence;i>0;i--)
3: i = 23
```

So there we have it! A poorly named loop variable that was re-used without permission. Calling the outer loop iCoinLoop (instead of the generic-looking *i*) would have raised eyebrows had it been erroneously reused here. This is a good advert for sensible naming conventions, as well as the power of watchpoints.

Had we missed this case, or we want to run the loop again to see how it got to this state, we can change the value of the variable with the *set* command:

```
(gdb) set var i = 0
```

Alternatively, we could call the function directly (with parameters). If there are no parameters, then you must still include the brackets.

```
(gdb) call CalcChangeFor(20)
```

This is our final fix. We re-compile, rerun, and hey presto – we have results!

```
$ change 23
1 x 20p
1 x 2p
```

Hmmm. I think we're being short changed! Try some other numbers. Do you notice a pattern?

The Logical Song

When you've found a bug, it is a very good idea to try and narrow the cases in which the bug occurs. Test the limits of your program: do negative numbers works? Odd numbers? Even numbers? Very big numbers? Very small numbers? The more case studies you have, the clearer the problem will become, and the less code that may be at fault.

In this case, all odd numbers exhibit the problem. This means it's probably an issue with the routine that deals out the pennies. If we don't wish to step around the loop 8 times, we can place a conditional breakpoint that will only fire when we're dealing with pennies.

```
(gdb) b 16 if val==1
Breakpoint 5 at 0x804855e: ➮
file change.c, line 16.
```

Because the debugger stops the program *before* the requested line is executed, we must place the breakpoint on the line after the assignment so that *val* will have been assigned correctly. Also note the C-style double equals in the condition. If something appears to go wrong, check your breakpoint logic – you might have a bug in your debugging method!

So, with breakpoint set we run the program and…nothing! The breakpoint doesn't fire. So the program never considers pennies. That would explain why we're getting short changed, but not how it happens. So, we run it again, this

Listing 5: Viewing the stack

```
(gdb) backtrace
#1  0x08048505 in CalcChangeFor (pence=23) at change.c:19
#2  0x08048505 in CalcChangeFor (pence=23) at change.c:19
#3  0x08048505 in CalcChangeFor (pence=23) at change.c:19
#4  0x08048505 in CalcChangeFor (pence=23) at change.c:19
....
```

Box 3: Numbers

Breakpoints are always numbered consecutively, even after a delete. So, if you start with 2 breakpoints (1 and 2). Issue 'delete'. The next breakpoint added will be numbered 3! This stops you from having to un-learn breakpoint assignments. This is also true of display variables and watchpoints.

time setting the breakpoint for the twopence pieces.

```
(gdb) b 16 if val==2
Note: breakpoint 5 also set
at pc 0x804855e.
Breakpoint 6 at 0x804855e:
file change.c, line 16.
```

This time the second breakpoint fires, and we can watch the debugger taking care of the tuppenny pieces. However, we also notice the loop doesn't continue with the next iteration. Printing out the variables in use will show us that the loop has legally terminated after 8 iterations. Checking the array length we see that there are, in fact, 9 different coin denominations! Such mistakes often occur with one line changes to 'just add support for 5 pound notes', without proper testing. Our finished masterpiece is now complete, as in Listing 7.

Instead of recompiling the program without debugging information, it is possible to strip out the debugging information from the existing executable with the command:

```
strip -g change
```

Split Decision

In large programs, adding breakpoints for every iteration of the loop is prohibitive. Imagine having to step through 1000 lines, to check the behaviour of the code. It is not necessary, however, to step through each one in turn, but to employ a technique known as the binary split.

Say we have narrowed our bug down to "somewhere in the program", we obviously want to narrow the potential area. The binary split poses the question, "Which half causes the bug? The first half, or the second half?"

To answer this, we place a breakpoint after the first half of the code and run it. If the problem has not yet manifested itself, then it is likely to be a fault with the last half. From here, we can ask the question again. Reducing the area under test to the 1st or 2nd quarter. This question can then be asked repeatedly until we're down to just one line, or a sufficiently small routine that we can step through line-by-line, studying its effects in more detail. A binary split can limit the search area of a 1,000 line program to just 10 steps! So, no matter how many millions of lines of code are in your DNA-based dino-park, resolving the bug is not as difficult as it may appear!

The Days of Pearly Spencer

Of course, the binary split, along with all other techniques presented here will not work in all cases. Debugging is as much a case of skill, perception and instinct, as it is about knowledge. But with a good command of the tools, and how to apply them, half the race is almost run. ■

Listing 6: Added breakpoints

```
(gdb) b 15
Breakpoint 2 at 0x8048554:
file change.c, line 15.
(gdb) r
Starting program: /home/steev
/code/change 23

Breakpoint 2, CalcChangeFor
(pence=23) at change.c:15
15          val = coins[i];
```

INFO

http://www.gnu.org/manual/gdb-5.1.1/gdb.html

http://systems.cs.colorado.edu/grunwald/Networks-spr01/gdb-refcard-letter.pdf

Listing 7: The final countdown

```
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  void CalcChangeFor(int pence)
 5  {
 6  int i, j, val;
 7  int coins[] = { 500,200,100,50,20,10,5,2,1,};
 8
 9          if (pence == 0)
10                  return; /* amount is exact */
11          if (pence < 0)
12                  return; /* invalid amount */
13
14          for(i=0;i<sizeof(coins)/sizeof(coins[0]);i++) {      /*
for each coin */
15                  val = coins[i];
16                  for(j=pence;j>0;j--)
17                          if (val*j <= pence) {
18                                  printf("%d x %dp\n", j, val);
19                                  CalcChangeFor(pence - j*val);
20                                  return;
21                                  }
22                  }
23  }
24
25  void usage(void)
26  {
27          fprintf(stderr, "Usage: change <amount>\n");
28  }
29
30  int main(int argc, char **argv)
31  {
32          if (argc > 1)
33                  CalcChangeFor(atoi(argv[1]));
34          else
35                  usage();
36          return 0;
37  }
```