A

Other Development Tools

Developing correct, fast C or C++ GNU/Linux programs requires more than just understanding the GNU/Linux operating system and its system calls. In this appendix, we discuss development tools to find runtime errors such as illegal use of dynamically allocated memory and to determine which parts of a program are taking most of the execution time. Analyzing a program's source code can reveal some of this information; by using these runtime tools and actually executing the program, you can find out much more.

A.1 Static Program Analysis

Some programming errors can be detected using static analysis tools that analyze the program's source code. If you invoke GCC with `-Wall` and `-pedantic`, the compiler issues warnings about risky or possibly erroneous programming constructions. By eliminating such constructions, you'll reduce the risk of program bugs, and you'll find it easier to compile your programs on different GNU/Linux variants and even on other operating systems.

Using various command options, you can cause GCC to issue warnings about many different types of questionable programming constructs. The `-Wall` option enables most of these checks. For example, the compiler will produce a warning about a comment that begins within another comment, about an incorrect return type specified for `main`, and about a non void function omitting a `return` statement. If you specify the `-pedantic` option, GCC emits warnings demanded by strict ANSI C and ISO C++ compliance. For example, use of the GNU `asm` extension causes a warning using this option. A few GNU extensions, such as using alternate keywords beginning with `__` (two underscores), will not trigger warning messages. Although the GCC info pages deprecate use of this option, we recommend that you use it anyway and avoid most GNU language extensions because GCC extensions tend to change through time and frequently interact poorly with code optimization.

Listing A.1 **(*hello.c*) Hello World Program**

```
main ()
{
  printf ("Hello, world.\n");
}
```

Consider compiling the "Hello World" program shown in Listing A.1. Though GCC compiles the program without complaint, the source code does not obey ANSI C rules. If you enable warnings by compiling with the `-Wall -pedantic`, GCC reveals three questionable constructs.

```
% gcc -Wall -pedantic hello.c
hello.c:2: warning: return type defaults to 'int'
hello.c: In function 'main':
hello.c:3: warning: implicit declaration of function 'printf'
hello.c:4: warning: control reaches end of non-void function
```

These warnings indicate that the following problems occurred:

- The return type for `main` was not specified.
- The function `printf` is implicitly declared because `<stdio.h>` is not included.
- The function, implicitly declared to return an `int`, actually returns no value.

Analyzing a program's source code cannot find all programming mistakes and inefficiencies. In the next section, we present four tools to find mistakes in using dynamically allocated memory. In the subsequent section, we show how to analyze the program's execution time using the `gprof` profiler.

A.2 Finding Dynamic Memory Errors

When writing a program, you frequently can't know how much memory the program will need when it runs. For example, a line read from a file at runtime might have any finite length. C and C++ programs use `malloc`, `free`, and their variants to dynamically allocate memory while the program is running. The rules for dynamic memory use include these:

- The number of allocation calls (calls to `malloc`) must exactly match the number of deallocation calls (calls to `free`).
- Reads and writes to the allocated memory must occur within the memory, not outside its range.
- The allocated memory cannot be used before it is allocated or after it is deallocated.

Because dynamic memory allocation and deallocation occur at runtime, static program analysis rarely find violations. Instead, memory-checking tools run the program, collecting data to determine if any of these rules have been violated. The violations a tool may find include the following:

- Reading from memory before allocating it
- Writing to memory before allocating it
- Reading before the beginning of allocated memory
- Writing before the beginning of allocated memory
- Reading after the end of allocated memory
- Writing after the end of allocated memory
- Reading from memory after its deallocation
- Writing to memory after its deallocation
- Failing to deallocate allocated memory
- Deallocating the same memory twice
- Deallocating memory that is not allocated

It is also useful to warn about requesting an allocation with 0 bytes, which probably indicates programmer error.

Table A.1 indicates four different tools' diagnostic capabilities. Unfortunately, no single tool diagnoses all the memory use errors. Also, no tool claims to detect reading or writing before allocating memory, but doing so will probably cause a segmentation fault. Deallocating memory twice will probably also cause a segmentation fault. These tools diagnose only errors that actually occur while the program is running. If you run the program with inputs that cause no memory to be allocated, the tools will indicate no memory errors. To test a program thoroughly, you must run the program using different inputs to ensure that every possible path through the program occurs. Also, you may use only one tool at a time, so you'll have to repeat testing with several tools to get the best error checking.

Table A.1 **Capabilities of Dynamic Memory-Checking Tools (X Indicates Detection, and O Indicates Detection for Some Cases)**

Erroneous Behavior	*malloc* Checking	mtrace	ccmalloc	Electric Fence
Read before allocating memory				
Write before allocating memory				
Read before beginning of allocation				X
Write before beginning of allocation	O		O	X
Read after end of allocation				X
Write after end of allocation			X	X
Read after deallocation				X
Write after deallocation				X
Failure to deallocate memory		X	X	
Deallocating memory twice	X		X	
Deallocating nonallocated memory		X	X	
Zero-size memory allocation			X	X

In the sections that follow, we first describe how to use the more easily used `malloc` checking and `mtrace`, and then `ccmalloc` and Electric Fence.

A.2.1 A Program to Test Memory Allocation and Deallocation

We'll use the `malloc-use` program in Listing A.2 to illustrate memory allocation, deallocation, and use. To begin running it, specify the maximum number of allocated memory regions as its only command-line argument. For example, `malloc-use 12` creates an array `A` with 12 character pointers that do not point to anything. The program accepts five different commands:

- To allocate *b* bytes pointed to by array entry `A[i]`, enter `a i b`. The array index *i* can be any non-negative number smaller than the command-line argument. The number of bytes must be non-negative.
- To deallocate memory at array index *i*, enter `d i`.
- To read the *p*th character from the allocated memory at index *i* (as in `A[i][p]`), enter `r i p`. Here, *p* can have an integral value.
- To write a character to the *p*th position in the allocated memory at index *i*, enter `w i p`.
- When finished, enter `q`.

We'll present the program's code later, in Section A.2.7, and illustrate how to use it.

A.2.2 *malloc* Checking

The memory allocation functions provided by the GNU C library can detect writing before the beginning of an allocation and deallocating the same allocation twice. Defining the environment variable `MALLOC_CHECK_` to the value 2 causes a program to halt when such an error is detected. (Note the environment variable's ending underscore.) There is no need to recompile the program.

We illustrate diagnosing a write to memory to a position just before the beginning of an allocation.

```
% export MALLOC_CHECK_=2
% ./malloc-use 12
Please enter a command: a 0 10
Please enter a command: w 0 -1
Please enter a command: d 0
Aborted (core dumped)
```

`export` turns on `malloc` checking. Specifying the value 2 causes the program to halt as soon as an error is detected.

Using `malloc` checking is advantageous because the program need not be recompiled, but its capability to diagnose errors is limited. Basically, it checks that the allocator data structures have not been corrupted. Thus, it can detect double deallocation of the same allocation. Also, writing just before the beginning of a memory allocation can usually be detected because the allocator stores the size of each memory allocation just before the allocated region. Thus, writing just before the allocated memory will corrupt this number. Unfortunately, consistency checking can occur only when your program calls allocation routines, not when it accesses memory, so many illegal reads and writes can occur before an error is detected. In the previous example, the illegal write was detected only when the allocated memory was deallocated.

A.2.3 Finding Memory Leaks Using *mtrace*

The `mtrace` tool helps diagnose the most common error when using dynamic memory: failure to match allocations and deallocations. There are four steps to using `mtrace`, which is available with the GNU C library:

1. Modify the source code to include `<mcheck.h>` and to invoke `mtrace ()` as soon as the program starts, at the beginning of `main`. The call to `mtrace` turns on tracking of memory allocations and deallocations.
2. Specify the name of a file to store information about all memory allocations and deallocations:

   ```
   % export MALLOC_TRACE=memory.log
   ```

3. Run the program. All memory allocations and deallocations are stored in the logging file.

4. Using the `mtrace` command, analyze the memory allocations and deallocations to ensure that they match.

```
% mtrace my_program $MALLOC_TRACE
```

The messages produced by `mtrace` are relatively easy to understand. For example, for our `malloc-use` example, the output would look like this:

```
- 0000000000 Free 3 was never alloc'd malloc-use.c:39

Memory not freed:
-----------------
   Address     Size     Caller
0x08049d48      0xc  at malloc-use.c:30
```

These messages indicate an attempt on line 39 of `malloc-use.c` to free memory that was never allocated, and an allocation of memory on line 30 that was never freed.

`mtrace` diagnoses errors by having the executable record all memory allocations and deallocations in the file specified by the `MALLOC_TRACE` environment variable. The executable must terminate normally for the data to be written. The `mtrace` command analyzes this file and lists unmatched allocations and deallocations.

A.2.4 Using *ccmalloc*

The `ccmalloc` library diagnoses dynamic memory errors by replacing `malloc` and `free` with code tracing their use. If the program terminates gracefully, it produces a report of memory leaks and other errors. The `ccmalloc` library was written by Armin Bierce.

You'll probably have to download and install the `ccmalloc` library yourself. Download it from `http://www.inf.ethz.ch/personal/biere/projects/ccmalloc/`, unpack the code, and run `configure`. Run `make` and `make install`, copy the `ccmalloc.cfg` file to the directory where you'll run the program you want to check, and rename the copy to `.ccmalloc`. Now you are ready to use the tool.

The program's object files must be linked with `ccmalloc`'s library and the dynamic linking library. Append `-lccmalloc -ldl` to your link command, for instance.

```
% gcc -g -Wall -pedantic malloc-use.o -o ccmalloc-use -lccmalloc -ldl
```

Execute the program to produce a report. For example, running our `malloc-use` program to allocate but not deallocate memory produces the following report:

```
% ./ccmalloc-use 12
file-name=a.out does not contain valid symbols
trying to find executable in current directory ...
using symbols from 'ccmalloc-use'
(to speed up this search specify 'file ccmalloc-use'
 in the startup file '.ccmalloc')
Please enter a command: a 0 12
Please enter a command: q
```

```
.--------------.
|ccmalloc report|
=======================================================
| total # of|   allocated | deallocated |     garbage |
+-----------+-------------+-------------+-------------+
|      bytes|          60 |          48 |          12 |
+-----------+-------------+-------------+-------------+
|allocations|           2 |           1 |           1 |
+-------------------------------------------------------+
| number of checks: 1                                   |
| number of counts: 3                                   |
| retrieving function names for addresses ... done.     |
| reading file info from gdb ... done.                  |
| sorting by number of not reclaimed bytes ... done.    |
| number of call chains: 1                              |
| number of ignored call chains: 0                      |
| number of reported call chains: 1                     |
| number of internal call chains: 1                     |
| number of library call chains: 0                      |
=======================================================
|
*100.0% = 12 Bytes of garbage allocated in 1 allocation
|       |
|       |       0x400389cb in <???>
|       |
|       |       0x08049198 in <main>
|       |                   at malloc-use.c:89
|       |
|       |       0x08048fdc in <allocate>
|       |                   at malloc-use.c:30
|       |
|       '-----> 0x08049647 in <malloc>
|                           at src/wrapper.c:284
|
'-------------------------------------------------------
```

The last few lines indicate the chain of function calls that allocated memory that was not deallocated.

To use `ccmalloc` to diagnose writes before the beginning or after the end of the allocated region, you'll have to modify the `.ccmalloc` file in the current directory. This file is read when the program starts execution.

A.2.5 Electric Fence

Written by Bruce Perens, Electric Fence halts executing programs on the exact line where a write or a read outside an allocation occurs. This is the only tool that discovers illegal reads. It is included in most GNU/Linux distributions, but the source code can be found at `http://www.perens.com/FreeSoftware/`.

As with `ccmalloc`, your program's object files must be linked with Electric Fence's library by appending `-lefence` to the linking command, for instance:

```
% gcc -g -Wall -pedantic malloc-use.o -o emalloc-use -lefence
```

As the program runs, allocated memory uses are checked for correctness. A violation causes a segmentation fault:

```
% ./emalloc-use 12
  Electric Fence 2.0.5 Copyright (C) 1987-1998 Bruce Perens.
Please enter a command: a 0 12
Please enter a command: r 0 12
Segmentation fault
```

Using a debugger, you can determine the context of the illegal action.

By default, Electric Fence diagnoses only accesses beyond the ends of allocations. To find accesses before the beginning of allocations *instead of* accesses beyond the end of allocations, use this code:

```
% export EF_PROTECT_BELOW=1
```

To find accesses to deallocated memory, set `EF_PROTECT_FREE` to 1. More capabilities are described in the `libefence` manual page.

Electric Fence diagnoses illegal memory accesses by storing each allocation on at least two memory pages. It places the allocation at the end of the first page; any access beyond the end of the allocation, on the second page, causes a segmentation fault. If you set `EF_PROTECT_BELOW` to 1, it places the allocation at the beginning of the second page instead. Because it allocates two memory pages per call to `malloc`, Electric Fence can use an enormous amount of memory. Use this library for debugging only.

A.2.6 Choosing Among the Different Memory-Debugging Tools

We have discussed four separate, incompatible tools to diagnose erroneous use of dynamic memory. How does a GNU/Linux programmer ensure that dynamic memory is correctly used? No tool guarantees diagnosing all errors, but using any of them does increase the probability of finding errors. To ease finding dynamically allocated memory errors, separately develop and test the code that deals with dynamic memory. This reduces the amount of code that you must search for errors. If you are using C++, write a class that handles all dynamic memory use. If you are using C, minimize the number of functions using allocation and deallocation. When testing this code, be sure to use only one tool at a one time because they are incompatible. When testing a program, be sure to vary how the program executes, to test the most commonly executed portions of the code.

Which of the four tools should you use? Because failing to match allocations and deallocations is the most common dynamic memory error, use `mtrace` during initial development. The program is available on all GNU/Linux systems and has been well tested. After ensuring that the number of allocations and deallocations match, use

Electric Fence to find illegal memory accesses. This will eliminate almost all memory errors. When using Electric Fence, you will need to be careful to not perform too many allocations and deallocations because each allocation requires at least two pages of memory. Using these two tools will reveal most memory errors.

A.2.7 Source Code for the Dynamic Memory Program

Listing A.2 shows the source code for a program illustrating dynamic memory allocation, deallocation, and use. See Section A.2.1, "A Program to Test Memory Allocation and Deallocation," for a description of how to use it.

Listing A.2 **(*malloc-use.c*) Dynamic Memory Allocation Checking Example**

```
/* Use C's dynamic memory allocation functions.  */

/* Invoke the program using one command-line argument specifying the
   size of an array.  This array consists of pointers to (possibly)
   allocated arrays.

   When the programming is running, select among the following
   commands:

   o allocate memory:   a <index> <memory-size>
   o deallocate memory: d <index>
   o read from memory:  r <index> <position-within-allocation>
   o write to memory:   w <index> <position-within-allocation>
   o quit:              q

   The user is responsible for obeying (or disobeying) the rules on dynamic
   memory use.  */

#ifdef MTRACE
#include <mcheck.h>
#endif /* MTRACE */
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

/* Allocate memory with the specified size, returning nonzero upon
   success.  */

void allocate (char** array, size_t size)
{
  *array = malloc (size);
}

/* Deallocate memory.  */

void deallocate (char** array)
```

continues

Listing A.2 **Continued**

```
{
  free ((void*) *array);
}

/* Read from a position in memory.  */

void read_from_memory (char* array, int position)
{
  char character = array[position];
}

/* Write to a position in memory.  */

void write_to_memory (char* array, int position)
{
  array[position] = 'a';
}

int main (int argc, char* argv[])
{
  char** array;
  unsigned array_size;
  char command[32];
  unsigned array_index;
  char command_letter;
  int size_or_position;
  int error = 0;

#ifdef MTRACE
  mtrace ();
#endif /* MTRACE */

  if (argc != 2) {
    fprintf (stderr, "%s: array-size\n", argv[0]);
    return 1;
  }

  array_size = strtoul (argv[1], 0, 0);
  array = (char **) calloc (array_size, sizeof (char *));
  assert (array != 0);

  /* Follow the user's commands.  */
  while (!error) {
    printf ("Please enter a command: ");
    command_letter = getchar ();
    assert (command_letter != EOF);
    switch (command_letter) {

    case 'a':
      fgets (command, sizeof (command), stdin);
      if (sscanf (command, "%u %i", &array_index, &size_or_position) == 2
          && array_index < array_size)
```

```
          allocate (&(array[array_index]), size_or_position);
        else
          error = 1;
        break;

      case 'd':
        fgets (command, sizeof (command), stdin);
        if (sscanf (command, "%u", &array_index) == 1
            && array_index < array_size)
          deallocate (&(array[array_index]));
        else
          error = 1;
        break;

      case 'r':
        fgets (command, sizeof (command), stdin);
        if (sscanf (command, "%u %i", &array_index, &size_or_position) == 2
            && array_index < array_size)
          read_from_memory (array[array_index], size_or_position);
        else
          error = 1;
        break;

      case 'w':
        fgets (command, sizeof (command), stdin);
        if (sscanf (command, "%u %i", &array_index, &size_or_position) == 2
            && array_index < array_size)
          write_to_memory (array[array_index], size_or_position);
        else
          error = 1;
        break;

      case 'q':
        free ((void *) array);
        return 0;

      default:
        error = 1;
      }
    }

    free ((void *) array);
    return 1;
  }
```

A.3 Profiling

Now that your program is (hopefully) correct, we turn to speeding its execution. Using the profiler `gprof`, you can determine which functions require the most execution time. This can help you determine which parts of the program to optimize or rewrite to execute more quickly. It can also help you find errors. For example, you may find that a particular function is called many more times than you expect.

In this section, we describe how to use `gprof`. Rewriting code to run more quickly requires creativity and careful choice of algorithms.

Obtaining profiling information requires three steps:

1. Compile and link your program to enable profiling.
2. Execute your program to generate profiling data.
3. Use `gprof` to analyze and display the profiling data.

Before we illustrate these steps, we introduce a large enough program to make profiling interesting.

A.3.1 A Simple Calculator

To illustrate profiling, we'll use a simple calculator program. To ensure that the calculator takes a nontrivial amount of time, we'll use unary numbers for calculations, something we would definitely not want to do in a real-world program. Code for this program appears at the end of this chapter.

A *unary number* is represented by as many symbols as its value. For example, the number 1 is represented by "x," 2 by "xx," and 3 by "xxx." Instead of using x's, our program represents a non-negative number using a linked list with as many elements as the number's value. The `number.c` file contains routines to create the number 0, add 1 to a number, subtract 1 from a number, and add, subtract, and multiply numbers. Another function converts a string holding a non-negative decimal number to a unary number, and a function converts from a unary number to an `int`. Addition is implemented using repeated addition of 1s, while subtraction uses repeated removal of 1s. Multiplication is defined using repeated addition. The unary predicates `even` and `odd` each return the unary number for 1 if and only if its one operand is even or odd, respectively; otherwise they return the unary number for 0. The two predicates are mutually recursive. For example, a number is even if it is zero, or if one less than the number is odd.

The calculator accepts one-line postfix expressions[1] and prints each expression's value—for example:

```
% ./calculator
Please enter a postfix expression:
2 3 +
5
Please enter a postfix expression:
2 3 + 4 -
1
```

1. In *postfix* notation, a binary operator is placed after its operands instead of between them. So, for example, to multiply 6 and 8, you would use 6 8 ×. To multiply 6 and 8 and then add 5 to the result, you would use 6 8 × 5 +.

The calculator, defined in `calculator.c`, reads each expression, storing intermediate values on a stack of unary numbers, defined in `stack.c`. The stack stores its unary numbers in a linked list.

A.3.2 Collecting Profiling Information

The first step in profiling a program is to annotate its executable to collect profiling information. To do so, use the `-pg` compiler flag when both compiling the object files and linking. For example, consider this code:

```
% gcc -pg -c -o calculator.o calculator.c
% gcc -pg -c -o stack.o stack.c
% gcc -pg -c -o number.o number.c
% gcc -pg calculator.o stack.o number.o -o calculator
```

This enables collecting information about function calls and timing information. To collect line-by-line use information, also specify the debugging flag `-g`. To count basic block executions, such as the number of `do`-loop iterations, use `-a`.

The second step is to run the program. While it is running, profiling data is collected into a file named `gmon.out`, only for those portions of the code that are exercised. You must vary the program's input or commands to exercise the code sections that you want to profile. The program must terminate normally for the profiling file to be written.

A.3.3 Displaying Profiling Data

Given the name of an executable, `gprof` analyzes the `gmon.out` file to display information about how much time each function required. For example, consider the "flat" profiling data for computing $1787 \times 13 - 1918$ using our calculator program, which is produced by executing `gprof ./calculator`:

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  ms/call  ms/call  name
 26.07      1.76     1.76 20795463     0.00     0.00  decrement_number
 24.44      3.41     1.65     1787     0.92     1.72  add
 19.85      4.75     1.34 62413059     0.00     0.00  zerop
 15.11      5.77     1.02     1792     0.57     2.05  destroy_number
 14.37      6.74     0.97 20795463     0.00     0.00  add_one
  0.15      6.75     0.01     1788     0.01     0.01  copy_number
  0.00      6.75     0.00     1792     0.00     0.00  make_zero
  0.00      6.75     0.00       11     0.00     0.00  empty_stack
```

Computing the function `decrement_number` and all the functions it calls required 26.07% of the program's total execution time. It was called 20,795,463 times. Each individual execution required 0.0 seconds—namely, a time too small to measure. The `add` function was invoked 1,787 times, presumably to compute the product. Each call

required 0.92 seconds. The `copy_number` function was invoked only 1,788 times, while it and the functions it calls required only 0.15% of the total execution time. Sometimes the `mcount` and `profil` functions used by profiling appear in the data.

In addition to the *flat profile data*, which indicates the total time spent within each function, `gprof` produces *call graph data* showing the time spent in each function and its children within the context of a function call chain:

```
index % time    self  children    called     name
                                                <spontaneous>
[1]    100.0    0.00    6.75                 main [1]
                0.00    6.75       2/2           apply_binary_function [2]
                0.00    0.00       1/1792        destroy_number [4]
                0.00    0.00       1/1           number_to_unsigned_int [10]
                0.00    0.00       3/3           string_to_number [12]
                0.00    0.00       3/5           push_stack [16]
                0.00    0.00       1/1           create_stack [18]
                0.00    0.00       1/11          empty_stack [14]
                0.00    0.00       1/5           pop_stack [15]
                0.00    0.00       1/1           clear_stack [17]
-----------------------------------------------
                0.00    6.75       2/2           main [1]
[2]    100.0    0.00    6.75       2         apply_binary_function [2]
                0.00    6.74       1/1           product [3]
                0.00    0.01       4/1792        destroy_number [4]
                0.00    0.00       1/1           subtract [11]
                0.00    0.00       4/11          empty_stack [14]
                0.00    0.00       4/5           pop_stack [15]
                0.00    0.00       2/5           push_stack [16]
-----------------------------------------------
                0.00    6.74       1/1           apply_binary_function [2]
[3]     99.8    0.00    6.74       1         product [3]
                1.02    2.65    1787/1792        destroy_number [4]
                1.65    1.43    1787/1787        add [5]
                0.00    0.00    1788/62413059     zerop [7]
                0.00    0.00       1/1792        make_zero [13]
```

The first frame shows that executing `main` and its children required 100% of the program's 6.75 seconds. It called `apply_binary_function` twice, which was called a total of two times throughout the entire program. Its caller was `<spontaneous>`; this indicates that the profiler was not capable of determining who called `main`. This first frame also shows that `string_to_number` called `push_stack` three times but was called five times throughout the program. The third frame shows that executing `product` and the functions it calls required 99.8% of the program's total execution time. It was invoked once by `apply_binary_function`.

The call graph data displays the total time spent executing a function and its children. If the function call graph is a tree, this number is easy to compute, but recursively defined functions must be treated specially. For example, the `even` function calls `odd`, which calls `even`. Each largest such call cycle is given its own number and is dis-

played individually in the call graph data. Consider this profiling data from determining whether $1787 \times 13 \times 3$ is even:

```
-----------------------------------------------
                0.00    0.02       1/1           main [1]
[9]      0.1    0.00    0.02       1         apply_unary_function [9]
                0.01    0.00       1/1           even <cycle 1> [13]
                0.00    0.00       1/1806        destroy_number [5]
                0.00    0.00       1/13          empty_stack [17]
                0.00    0.00       1/6           pop_stack [18]
                0.00    0.00       1/6           push_stack [19]
-----------------------------------------------
[10]     0.1    0.01    0.00       1+69693   <cycle 1 as a whole> [10]
                0.00    0.00   34847             even <cycle 1> [13]
-----------------------------------------------
                               34847             even <cycle 1> [13]
[11]     0.1    0.01    0.00   34847         odd <cycle 1> [11]
                0.00    0.00   34847/186997954     zerop [7]
                0.00    0.00       1/1806        make_zero [16]
                               34846             even <cycle 1> [13]
```

The `1+69693` in the `[10]` frame indicates that cycle 1 was called once, while the functions in the cycle were called 69,693 times. The cycle called the `even` function. The next entry shows that `odd` was called 34,847 times by `even`.

In this section, we have briefly discussed only some of `gprof`'s features. Its info pages contain information about other useful features:

- Use the `-s` option to sum the execution results from several different runs.
- Use the `-c` option to identify children that could have been called but were not.
- Use the `-l` option to display line-by-line profiling information.
- Use the `-A` option to display source code annotated with percentage execution numbers.

The info pages also provide more information about the interpretation of the analyzed data.

A.3.4 How *gprof* Collects Data

When a profiled executable runs, every time a function is called its count is also incremented. Also, `gprof` periodically interrupts the executable to determine the currently executing function. These samples determine function execution times. Because Linux's clock ticks are 0.01 seconds apart, these interruptions occur, at most, every 0.01 seconds. Thus, profiles for quickly executing programs or for quickly executing infrequently called functions may be inaccurate. To avoid these inaccuracies, run the executable for longer periods of time, or sum together profile data from several executions. Read about the `-s` option to sum profiling data in `gprof`'s info pages.

A.3.5 Source Code for the Calculator Program

Listing A.3 presents a program that calculates the value of postfix expressions.

Listing A.3 **(*calculator.c*) Main Calculator Program**

```
/* Calculate using unary numbers.  */

/* Enter one-line expressions using reverse postfix notation, e.g.,
     602 7 5 - 3 * +
   Nonnegative numbers are entered using decimal notation.  The
   operators "+", "-", and "*" are supported.  The unary operators
   "even" and "odd" return the number 1 if its one operand is even
   or odd, respectively.  Spaces must separate all words.  Negative
   numbers are not supported.  */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "definitions.h"


/* Apply the binary function with operands obtained from the stack,
   pushing the answer on the stack.  Return nonzero upon success.  */

int apply_binary_function (number (*function) (number, number),
                           Stack* stack)
{
  number operand1, operand2;
  if (empty_stack (*stack))
    return 0;
  operand2 = pop_stack (stack);
  if (empty_stack (*stack))
    return 0;
  operand1 = pop_stack (stack);
  push_stack (stack, (*function) (operand1, operand2));
  destroy_number (operand1);
  destroy_number (operand2);
  return 1;
}

/* Apply the unary function with an operand obtained from the stack,
   pushing the answer on the stack.  Return nonzero upon success.  */

int apply_unary_function (number (*function) (number),
                          Stack* stack)
{
  number operand;
  if (empty_stack (*stack))
    return 0;
```

```
  operand = pop_stack (stack);
  push_stack (stack, (*function) (operand));
  destroy_number (operand);
  return 1;
}

int main ()
{
  char command_line[1000];
  char* command_to_parse;
  char* token;
  Stack number_stack = create_stack ();

  while (1) {
    printf ("Please enter a postfix expression:\n");
    command_to_parse = fgets (command_line, sizeof (command_line), stdin);
    if (command_to_parse == NULL)
      return 0;

    token = strtok (command_to_parse, " \t\n");
    command_to_parse = 0;
    while (token != 0) {
      if (isdigit (token[0]))
        push_stack (&number_stack, string_to_number (token));
      else if (((strcmp (token, "+") == 0) &&
                !apply_binary_function (&add, &number_stack)) ||
               ((strcmp (token, "-") == 0) &&
                !apply_binary_function (&subtract, &number_stack)) ||
               ((strcmp (token, "*") == 0) &&
                !apply_binary_function (&product, &number_stack)) ||
               ((strcmp (token, "even") == 0) &&
                !apply_unary_function (&even, &number_stack)) ||
               ((strcmp (token, "odd") == 0) &&
                !apply_unary_function (&odd, &number_stack)))
        return 1;
      token = strtok (command_to_parse, " \t\n");
    }
    if (empty_stack (number_stack))
      return 1;
    else {
      number answer = pop_stack (&number_stack);
      printf ("%u\n", number_to_unsigned_int (answer));
      destroy_number (answer);
      clear_stack (&number_stack);
    }
  }

  return 0;
}
```

The functions in Listing A.4 implement unary numbers using empty linked lists.

Listing A.4 **(*number.c*) Unary Number Implementation**

```
/* Operate on unary numbers.  */

#include <assert.h>
#include <stdlib.h>
#include <limits.h>
#include "definitions.h"

/* Create a number representing zero.  */

number make_zero ()
{
  return 0;
}

/* Return nonzero if the number represents zero.  */

int zerop (number n)
{
  return n == 0;
}

/* Decrease a positive number by 1.  */

number decrement_number (number n)
{
  number answer;
  assert (!zerop (n));
  answer = n->one_less_;
  free (n);
  return answer;
}

/* Add 1 to a number.  */

number add_one (number n)
{
  number answer = malloc (sizeof (struct LinkedListNumber));
  answer->one_less_ = n;
  return answer;
}

/* Destroying a number.  */

void destroy_number (number n)
{
  while (!zerop (n))
    n = decrement_number (n);
```

```
}

/* Copy a number.  This function is needed only because of memory
   allocation.  */

number copy_number (number n)
{
  number answer = make_zero ();
  while (!zerop (n)) {
    answer = add_one (answer);
    n = n->one_less_;
  }
  return answer;
}

/* Add two numbers.  */

number add (number n1, number n2)
{
  number answer = copy_number (n2);
  number addend = n1;
  while (!zerop (addend)) {
    answer = add_one (answer);
    addend = addend->one_less_;
  }
  return answer;
}

/* Subtract a number from another.  */

number subtract (number n1, number n2)
{
  number answer = copy_number (n1);
  number subtrahend = n2;
  while (!zerop (subtrahend)) {
    assert (!zerop (answer));
    answer = decrement_number (answer);
    subtrahend = subtrahend->one_less_;
  }
  return answer;
}

/* Return the product of two numbers.  */

number product (number n1, number n2)
{
  number answer = make_zero ();
  number multiplicand = n1;
  while (!zerop (multiplicand)) {
    number answer2 = add (answer, n2);
    destroy_number (answer);
```

continues

Listing A.4 **Continued**

```
    answer = answer2;
    multiplicand = multiplicand->one_less_;
  }
  return answer;
}

/* Return nonzero if number is even.  */

number even (number n)
{
  if (zerop (n))
    return add_one (make_zero ());
  else
    return odd (n->one_less_);
}

/* Return nonzero if number is odd.  */

number odd (number n)
{
  if (zerop (n))
    return make_zero ();
  else
    return even (n->one_less_);
}

/* Convert a string representing a decimal integer into a "number".  */

number string_to_number (char * char_number)
{
  number answer = make_zero ();
  int num = strtoul (char_number, (char **) 0, 0);
  while (num != 0) {
    answer = add_one (answer);
    --num;
  }
  return answer;
}

/* Convert a "number" into an "unsigned int".  */

unsigned number_to_unsigned_int (number n)
{
  unsigned answer = 0;
  while (!zerop (n)) {
    n = n->one_less_;
    ++answer;
  }
  return answer;
}
```

The functions in Listing A.5 implement a stack of unary numbers using a linked list.

Listing A.5 **(*stack.c*) Unary Number Stack**

```
/* Provide a stack of "number"s.  */

#include <assert.h>
#include <stdlib.h>
#include "definitions.h"

/* Create an empty stack.  */

Stack create_stack ()
{
  return 0;
}

/* Return nonzero if the stack is empty.  */

int empty_stack (Stack stack)
{
  return stack == 0;
}

/* Remove the number at the top of a nonempty stack.  If the stack is
   empty, abort.  */

number pop_stack (Stack* stack)
{
  number answer;
  Stack rest_of_stack;

  assert (!empty_stack (*stack));
  answer = (*stack)->element_;
  rest_of_stack = (*stack)->next_;
  free (*stack);
  *stack = rest_of_stack;
  return answer;
}

/* Add a number to the beginning of a stack.  */

void push_stack (Stack* stack, number n)
{
  Stack new_stack = malloc (sizeof (struct StackElement));
  new_stack->element_ = n;
  new_stack->next_ = *stack;
  *stack = new_stack;
}

/* Remove all the stack's elements.  */
```

continues

Listing A.5 **Continued**

```
void clear_stack (Stack* stack)
{
  while (!empty_stack (*stack)) {
    number top = pop_stack (stack);
    destroy_number (top);
  }
}
```

Listing A.6 contains declarations for stacks and numbers.

Listing A.6 **(*definitions.h*) Header File for number.c and stack.c**

```
#ifndef DEFINITIONS_H
#define DEFINITIONS_H 1

/* Implement a number using a linked list.  */
struct LinkedListNumber
{
  struct LinkedListNumber*
                one_less_;
};
typedef struct LinkedListNumber* number;

/* Implement a stack of numbers as a linked list.  Use 0 to represent
   an empty stack.  */
struct StackElement
{
  number        element_;
  struct        StackElement* next_;
};
typedef struct StackElement* Stack;

/* Operate on the stack of numbers.  */
Stack create_stack ();
int empty_stack (Stack stack);
number pop_stack (Stack* stack);
void push_stack (Stack* stack, number n);
void clear_stack (Stack* stack);

/* Operations on numbers.  */
number make_zero ();
void destroy_number (number n);
number add (number n1, number n2);
number subtract (number n1, number n2);
number product (number n1, number n2);
number even (number n);
number odd (number n);
number string_to_number (char* char_number);
unsigned number_to_unsigned_int (number n);

#endif /* DEFINITIONS_H */
```