14

14 Tools

This chapter introduces two powerful support tools.

14.1 THE "CODE COVERAGE" TOOL

Role of a code coverage tool

A "code coverage" tool helps you determine whether you have properly tested your program. Think about all the `if … else …` and `switch()` statements in your code. Each involves optionally executed code, with calls to other functions where there are further conditional constructs. These conditional constructs mean that there can be a very large number of different execution paths through your code.

You can never be certain that you have tested all paths through a complex program; so there will always be chances that some paths have errors where data are combined incorrectly. But if you have never tested certain paths, there can be gross errors that will crash your program.

The simple programs that we have considered so far don't really justify the use of a code coverage tool; after all, things simulating the π-cannon don't involve that many choices and there is very little dependence on different inputs. But as you move to more complex things, like some of the "tree algorithms" in Part IV, code coverage tools become more and more necessary. The "tree algorithms" build up complicated data structures that depend on the input data. Rules implemented in the algorithms define how these structures are to change as new data elements get added. Some of these rules apply to relatively rare special cases that occur only for specific combinations of data values. The code for the "tree algorithms" has numerous paths, including those for these rare special cases.

How can you test all paths? Basically, you have to chose different input data so as to force all processing options to be executed. Choosing the data is sometimes hard; you may think that you have included examples that cover all cases, but it is difficult to be certain.

This is where code coverage tools get involved. The basic idea is that the compiler and run-time system should help you check that your tests have at least executed the

code on every conditional branch of your program. Unfortunately, a "code coverage" tool is not yet a standard part of the IDEs available for personal computers. The example in this section uses the "tcov" tool on Sun Unix.

Compiler adds record keeping code

Special compile time options have to be specified when preparing a program for this form of analysis. These options direct the compiler to add extra instructions to those that would be generated from the program's source code. As well as extra instructions embedded in the code, the compiler adds an extra data table, a startup function and a termination function. The extra instructions, and related components, arrange for counts to be kept of the number of times that each conditional block is executed, when the program terminates these counts are saved to a data file.

You can imagine the process as being one where the compiler converts the following simple program:

```
#include <iostream.h>

int main()
{
    cout << "Enter Number ";
    int num;
    cin >> num;
    if(num >= 0)
           cout << "that was positive" << endl;
    else
           cout << "that was negative" << endl;
    return 0;
}
```

into an "instrumented" version:

```
#include <iostream.h>

int __counters[3];
extern void __InitializeCounters(int d[], int n);
extern void __SaveCountersToFile(int d[], int n);

int main()
{
    __InitializeCounters(__counters, 3);
    __counters[0]++;
    cout << "Enter Number ";
    int num;
    cin >> num;
    if(num >= 0) {
           __counters[1]++;
           cout << "that was positive" << endl;
           }
    else   {
           __counters[2]++;
           cout << "that was negative" << endl;
```

```
            }
      __SaveCountersToFile(__counters, 3);
      return 0;
}
```

The "instrumented" program will run just like the original, except that it saves the counts to a file before terminating. In a properly implemented system, each time the program is run the new counts are added to those in any existing record file.

Analysis tool

When you have run your program several times on different data, you get the analysis tool to interpret the contents of the file with the counts. This analysis tool combines the counts data with the original source code to produce a listing that illustrates the number of times each different conditional block of code has been executed. Sections of code that have never been executed are flagged, warning the tester that checks are incomplete. The following output from the Unix tcov tool run on a version of the Quicksort program from Chapter 13.

```
            const int kSMALL_ENOUGH = 15;
            const int kBIG = 50000;
            int data[kBIG];

            void SelectionSort(int data[], int left, int right)
     6246 -> {
                    for(int i = left; i < right; i++) {
     43754 ->               int min = i;
                            for(int j=i+1; j<= right; j++)
    248487 ->                       if(data[j] < data[min]) min =
     35251 ->               int temp = data[min];
     43754 ->               data[min] = data[i];
                            data[i] = temp;
                    }
     6246 -> }

            int Partition( int d[], int left, int right)
     6245 -> {
                    int val =d[left];
                    int lm = left-1;
                    int rm = right+1;
                    for(;;) {
    152418 ->               do
                                    rm--;
    518367 ->               while (d[rm] > val);
    152418 ->               do
                                    lm++;
    412418 ->               while( d[lm] < val);
    152418 ->               if(lm<rm) {
    146173 ->                       int tempr = d[rm];
                                    d[rm] = d[lm];
                                    d[lm] = tempr;
                                    }
```

```
                        else
 6245 ->                        return rm;
##### ->                }
##### -> }

         void Quicksort( int d[], int left, int right)
12491 -> {
                if(left < (right-kSMALL_ENOUGH)) {
 6245 ->                int split_pt = Partition(d,left,
                        Quicksort(d, left, split_pt);
                        Quicksort(d, split_pt+1, right);
                        }
 6246 ->        else SelectionSort(d, left, right);
 6246 -> }

         int main()
    1 -> {
                int i;
                long sum = 0;
                for(i=0;i <kBIG;i++)
50000 ->                sum += data[i] = rand() % 15000;

50000 ->        Quicksort(data, 0, kBIG-1);

    1 ->        int last = -1;
                long sum2 = 0;
                for(i = 0; i < kBIG; i++)
50000 ->                if(data[i] < last) {
##### ->                        cout << "Oh ....; the data
                               cout << "Noticed the problem at
                               exit(1);
                               }
50000 ->                else {
                               sum2 += last = data[i];
                               }
50000 ->        if(sum != sum2) {
##### ->                cout << "Oh ...; we seem to have
                        }

    1 ->        return 0;
```

All the parts of the program that we want executed have indeed been executed.

For real programs, code coverage tools are an essential part of the test and development process.

14.2 THE PROFILER

A code coverage tool helps you check that you have tested your code, a "Profiler" helps you make your code faster.

First some general advice on speeding up programs:

General advice 1:

1. Make it work.
2. Make it fast.

Step 2 is optional.

General advice 2:

The slow bit isn't the bit you thought would be slow.

General advice 3:

Fiddling with "register" declarations, changing from arrays to pointers and general hacking usually makes not one measurable bit of difference.

General advice 4

The only way to speed something up significantly will involve a fundamental change of algorithm (and associated data structure).

General advice 5

Don't even think about making changes to algorithms until you have acquired some detailed timing statistics.

Most environments provide some form of "profiling" tool that can be used to get the timing statistics that show where a program really is spending its time. To use a profiling tool, a program has to be compiled with special compile-time flags set. When these flags are set, the compiler and link-loader set up auxiliary data structures that allow run time addresses to be mapped back to source code. They also arrange for some run-time component that can be thought of as regularly interrupting a running program and asking where its program counter (PC) is. This run-time component takes the PC value and works out which function it corresponds to and updates a counter for that function. When the program finishes, the counts (and mapping data) are saved to file. The higher the count associated with a particular bit of code, the greater the time spent in that code.

A separate utility program can take the file with counts, and the source files and produce a summary identifying which parts of a program use most time. The following data were obtained by using the Unix profiler to analyze the performance of the extended Quicksort program:

```
%Time Seconds Cumsecs     Name
 44.7    0.42    0.42     __0FJPartitionPiiTC
 23.4    0.22    0.64     __0FNSelectionSortPiiTC
 11.7    0.11    0.75     main
  6.4    0.06    0.81    .rem
  6.4    0.06    0.87    .mul
  1.1    0.01    0.94     rand
```

Both the Symantec and the Borland IDEs include profilers that can provide information to that illustrated.

The largest portion of this program's time is spent in the `Partition()` function, the `Quicksort()` function itself doesn't even register because it represents less than 1% of the time. (The odd names, like `__0FNSelectionSortPiiTC` are examples of "mangled" names produced by a C++ compiler.)

Once you know where the program spends its time, then you can start thinking about minor speed ups (register declarations etc) or major improvements (better algorithms).