

The Application of C Programming Techniques

Ryan Botzler
John Carroll School
August 11, 2000
ARL
Frederick S. Brundick

ABSTRACT

This report documents the many different software development tools needed to combine the Bossert-Hergert encoder and decoder by using the C programming language. The combination of the two programs enables a user to perform the encoding and decoding routines while only dealing with a single program. The Bossert-Hergert software allows the user to encode a message into binary code and add error correction code, so that the message may be later decoded back, while utilizing the error correction code to reduce the number of errors in the message.

1. INTRODUCTION

1.1 Purpose

The Bossert-Hergert software is a way to allow a message to be encoded into a binary file and inserts error correction code, so that the result may be decoded later, using the inserted error correction code to eliminate any errors. The combination of the encoding and decoding processes allows the user to deal with only one program for both functions. Combining the two programs required searching through both programs for common steps and code. Once I found common variables and functions, I was able to merge the two programs into one program. Combining these two programs made the processes more accessible and allowed for later alterations to enhance the program.

1.2 Background

The Bossert-Hergert software utilizes low rate Reed-Solomon codes. Reed-Solomon codes are error-correcting codes that are well-understood and widely used. After research and calculations, the trillions of codes were narrowed down to only four sets of effective codewords. These sets had the parameters (31,7), (63,6), (127,5), and (255,4). The codewords read the encoded information and perform error correction routines during the decoding of the message.

For each codeword set, there are two files that are used: the G-file and the H-file. The G-file, which is used for the encoding process, inserts error-correction code into the message. The H-file, which is used for decoding, utilizes the error-correction code to eliminate the errors while decoding the message. Currently, there are four sets of G-files and H-files, one set for each of the four effective sets of codewords.

2. UNIX SOFTWARE

2.1 gcc Compiler

The **gcc** compiler is a UNIX tool that allows a user to compile a C program into an object file and an executable file. The program that is to be compiled must be free from errors, or else the process will fail. **gcc** provided me with a sufficient list of warnings

and errors in the construction of the new files. The **gcc** compiler works in four main steps: the preprocessor, the compiler, the assembler, and the linker.

2.1.1 Preprocessor

The preprocessor resolves commands in the code like **#define**, **#include**, **#ifdef**, etc.. Similar to other UNIX systems, **gcc** calls a separate tool called **cpp** to carry out the preprocessing step.

2.1.2 Compiler

In the compiling stage, the software produces assembly language, which is usually not saved, but is used in the assembly process.

2.1.3 Assembler

This process receives the assembly language created by the compilation step. The output of the assembly is an object file with a **.o** extension.

2.1.4 Linking

The final stage, the linking stage, takes the **.o** object files and arranges them properly to create an executable file. This output is now a file that is capable of running the program that has been compiled.

2.1.5 Parameters for **gcc**

gcc can stop at any stage in compiling the file, depending on different parameters added to the command line. Compiling the Bossert-Hergert source required many passes through the **gcc** compiler. In those passes to find errors and warnings, the parameter **-c** was added to the command line. This parameter stopped the compilation process after the assembly stage, which yields only an object file. Since some of the necessary functions were written in other programs, linking required including the object files of several other source files. Stopping the process early allowed me to avoid unnecessary errors that are due to undeclared functions and variables or other problems that may be resolved in the other programs.

Another parameter which I used in the compilation of the Bossert-Hergert software was **-Wall**. This parameter allowed the user to see all of the warnings that occur. This feature is useful to try to eliminate all of the possible warnings that may infringe on the way the program runs. Also, these warnings allow the user to see possible typographical errors in writing the program that may not be otherwise detected.

2.2 diff Text Comparison

diff is a program that compares two text files. After a message was encoded and then decoded, **diff** was important in comparing the two documents. Since there may be errors from decoding the message from the binary file, the two documents may differ slightly. **diff** was able to locate the differences in the texts by comparing the two documents side by side. This tool was used to insure that the encoding and decoding worked properly.

2.3 vi Editor

Before I could start writing any code, I had to learn how to use the **vi** editor. **vi** is an full screen editor commonly used in UNIX, but I used it on a Windows-based PC system. **vi** had a system of commands for me to learn. Such commands were necessary in the writing of the code for the combined Bossert-Hergert software.

2.4 RCS Source Management

The Revision Control System, or RCS, is a management tool used in UNIX. RCS allows editing source code programs while keeping the original version intact. This software uses the **diff** command to note how a file has changed. Also this software allows a version of a file to be edited by only one user at a time. This feature is very helpful in limiting the confusion when more than one person is working on a project. The saving of multiple versions allows users to retrieve files before the alterations were made to either start over or to branch off into another project. RCS saves the revisions in a tree structure, showing from where each version came. In the case of the Bossert-Hergert files, I could revise earlier editions of the combined program if I needed to retrieve lines of code before I made changes. RCS acts as a helpful piece of restoring software that enables the user to retain the prior versions of a file, in case of an error in future versions.

RCS requires the editor of a file to input a descriptive comment line for each version created. This comment line is very helpful in that it allows a description of the changes made to be written. The version descriptions create a history list of what has been done for each revision of a file. This is very useful when multiple users edit a program. The version comment lines allow the user to see what had changed and who modified the program. That way, the user may decide which version to edit, just by checking the list of revisions.

2.5 lint Checking

lint is a very useful tool for UNIX programming. Although compilers, such as **gcc**, remind the user of errors and warnings in the code while it is compiling, **lint** continues further. **lint** tests the code more for portability errors and checks the programs against standard libraries. **lint** is designed to create cleaner, more portable, and more effective code. The messages that **lint** returns are sometimes actual problems with the code, but

sometimes they are glitches that may be ignored by the user. **lint** helps programmers take a closer look at their programs, so that no warnings can hinder their efficiency.

2.6 make Management

make is a command generator used in large compilations of sets of programs. This software sorts out dependency relationships among files. **make** determines what needs to be created or compiled depending on other files by writing out dependencies in the **make** file. If one component of a large project is altered, then a **make** file can be written to build only what is needed. **make** can help coordinate contributions made to a group of programs by many different users.

3. PROJECT DESCRIPTION

Combining the two programs required searching through both programs for common steps and code. Once I found common variables and functions, I was able to merge the two programs into one program. The overlap was minimal between the two processes, so I took the functions of each of the programs and put them side by side, using the **vi** editor, into the combined program. The combined program needed a main function, as all programs need, that needed to determine which subroutine to run, encoding or decoding.

3.1 Parameters

When writing the main function, I learned that the arguments needed for determining which process to use were selected within the command line. I was introduced to the standard variables of **argc** and **argv[]**. **argc** stands for argument count and holds the value of the number of arguments inputted. Arguments include the name of the program and any parameters that may be added. **argv[]** is an array of pointers which stores the location of each of the arguments. Using these variables, I was able to create a selection process for the combined program.

The standard for command lines in DOS and UNIX includes the name of the program, the parameters passed to the program, then the files that are to be used in the process. The standard for parameters is to have a dash (-) in front of each parameter, but there are many ways to select the desired process. In order to determine which process the program will carry out, the program should accept a parameter of either **-e** or **-d**, to denote encoding and decoding respectively. This argument is required and must be one of the two processes, since the program is relatively simple. Each process had its set of additional parameters that were either optional or required for the running of the program. The encoding did not require any additional parameters, but could accept a verbose parameter, denoted by **-v**. Although the verbose flag is not used in the encoding yet, the program may be altered at a later time to include a verbose option within the subroutines. The decoding subroutines require the maximum number of steps (max steps) for decoding to be entered. It also can accept a verbose flag, which causes the program to print different lines of checks on the program's progress.

3.2 Error Messages

The program will fail if the incorrect parameters were given. Instead of just falling out of the program, it is more informative to include a simple explanation for why the program has failed. Since **-e** or **-d** is required, an error message occurs when either no arguments after the program name are specified or the user did not select either of them. Since a value for the max steps is required for the decode process, an error message appears, stating that the max steps value is absent. In the case of a parameter that is not defined, a separate error message occurs. The program also catches the case where too many arguments are passed and displays an error message.

3.3 Compilation

After the main function was written, the program was ready for **gcc** and **lint**. These two tools helped find the errors and warnings in the code. Since the program was divided among different files, linking was going to create a problem, so the **-c** parameter from **gcc** was added to the command line. The **-Wall** command in **gcc** was helpful in this step and allowed me to find many of the errors and warnings in the program. After many passes through this software and many small alterations to the program, I was able to eliminate all errors and minimize the number of warnings. At that point, the program was ready to be linked. In doing that, the names of all of the files that held code for the program were entered into the command line and the **-c** parameter was omitted. The program was compiled and linked into an executable file. This executable file was ready to run and test.

3.4 Testing

The newly formed executable file still needed to be tested. By testing the program, I ran it several times, omitting different parameters and checking to see what error messages occurred. Since I received the encode and decode programs from another programmer and the programs were functional, I assumed that each process was doing what it should and tested the program on a text file. The text file was successfully encoded into a binary file during the encoding process to start the test. Through a separate command, the newly created binary file was then sent through the program's decoding process and a text file was created. By running **diff**, I was able to see that the two documents seemed to be identical. This was a sign of successful passes through both of the processes. I continued with a few more passes with different text files to insure that the program was functioning.

3.5 make File Creation

I was able to create a simple **make** file that regulates the compilation of the software. The creation of a **make** file will be helpful in compiling future versions of the Bossert-Hergert software. As more code is added for the enhancements to the program, the number of files may increase. As the set of files grows larger and more complex, the

make file may prove to be beneficial in compiling the program. The **make** file would allow alterations to be done on some parts of the program while leaving the rest intact during compilation. In addition, this file will save time during compiling the program due to its efficiency.

4. FUTURE ENHANCEMENTS

4.1 Verbose Flag

Although the encoding routine currently allows a verbose flag to be inputted, the code for the encoding process did not use it. The verbose flag was mainly for checking the decoding of the message, but a verbose process may be later implemented into the encoding to check what the process is specifically doing and how it is running.

The verbose flag in the decoding process originally had multiple levels at which the program could be checked. To simplify the process of checking for the flag, I combined the multiple levels of verbose into one level which included all of the checks in each of the levels. In future projects which include this program, the code may be altered to include the multiple levels of verbose rather than just one.

4.2 Codeword Options

The program currently is set to using the (255,4) codeword set to encode or decode the error-correction information by using hard-coded constants. The program could be enhanced to receive one of the four different sets of codewords within the command line by a series of conditional statements and variable assignments. The program would also open the corresponding G-file or H-file, depending on which codeword set was selected. That way, the user can specify which set of codewords to use while encoding or decoding. The variable assignment choices will be set into the program, since only four codeword sets are used. Depending on which codeword set is specified, the program will alter its routines slightly to suit the set which was inputted.

5. CONCLUSIONS

The Bossert-Hergert software is a very useful application to use while encoding or decoding a message. The combination of the two processes will allow users to call a single program for both processes. The future extensions that may be added to this program will make this software even more complete and accessible. Combining this software will promote more wide-spread use by simplifying the user access.

The many tools used to combine the Bossert-Hergert encoder and decoder were very helpful utilities. Each played an important role in the construction of the combined program. These common tools are beneficial and efficient methods used to work on projects in a UNIX environment. They provide the user with the capability to perform more complex operations in his or her projects.

Bibliography

Darwin, Ian F.. Checking C Programs with **lint**. Sebastopol, California: O'Reilly & Associates, Inc., 1991.

Kernighan, Brian W. and Rob Pike. The UNIX Programming Environment. London: Prentice-Hall International, Inc., 1984.

Loukides, Mike and Andy Oram. Programming with GNU Software. Sebastopol, California: O'Reilly & Associates, Inc., 1997.

Marvel, Lisa M., Charles G. Boncelet, Jr., and Charles T. Retter. Methodology of Spread-Spectrum Image Steganography. Aberdeen Proving Ground, Maryland: Army Research Laboratory, 1998.

Oram, Andrew and Steve Talbott. Managing Projects with **make**. Sebastopol, California: O'Reilly & Associates, Inc., 1993.

Retter, Charles T.. Binary Weight Distributions of Low Rate Reed-Solomon Codes. Aberdeen Proving Ground, Maryland: Army Research Laboratory, 1995.