The purpose of this project was to design and implement a C language program reformatter. This program is able to accept one or more valid C programs as input and generate as output equivalent C program(s) in a book format representation. Book format is a typographic style consisting of a Title Page, a Table of Contents, the input program(s) reformatted for readability and understandability, a listing of any user-defined header files, and a Function Cross-Reference Index.
ACKNOWLEDGEMENTS

I would like to thank Dr. Patrick Michaud for providing the idea for this project. His guidance and encouragement helped make the completion of this project a reality.

I would also like to thank Dr. Herbert Haynes and Mr. David Perkins for agreeing to be on my committee and taking the time and effort to review this document. "Thank you" also to Dr. Roy Ellzey who has given me encouragement and support throughout my enrollment at Corpus Christi State University.

A special thanks to Ms. Joyce Purdy and Ms. Barbara Stewart of Hoechst Celanese Corporation. I could not have had two finer professional or personal role models while pursuing an advanced degree.
# TABLE OF CONTENTS

ABSTRACT .................................................. ii

ACKNOWLEDGEMENTS ........................................ iii

LIST OF FIGURES and TABLES ............................. ix

CHAPTER 1: INTRODUCTION ................................. 1
  1.1 BACKGROUND ........................................... 1
  1.2 PROJECT OVERVIEW ..................................... 3
  1.3 ENVIRONMENT ........................................... 5

CHAPTER 2: PROGRAM DESCRIPTION ....................... 6
  2.1 PROCEDURE ............................................. 6
  2.2 REFORMAT.C ............................................. 7
    2.2.1 Main ............................................... 7
    2.2.2 Build_Reserved_Table .............................. 10
  2.3 PARSER.C ............................................. 10
    2.3.1 Parse ............................................. 10
    2.3.2 Add_To_Struct_List ................................ 12
    2.3.3 Back_Track ........................................ 13
    2.3.4 Check_If_Function ................................ 13
    2.3.5 Check_If_Reserved ................................. 14
    2.3.6 Check_Line_Length ................................. 14
    2.3.7 End_Lib_Include ................................... 16
2.3.8  Enum_Cont .......................... 17
2.3.9  Finish_Struct ...................... 17
2.3.10 Get_Enum_List ..................... 18
2.3.11 Init_Struct ......................... 19
2.3.12 Invalid_Statement .................. 20
2.3.13 Next_Line .......................... 20
2.3.14 Next_Token ......................... 21
2.3.15 P_Case .............................. 22
2.3.16 P_Comment ......................... 26
2.3.17 P_Define ............................ 26
2.3.18 P_Enum .............................. 27
2.3.19 P_For ............................... 27
2.3.20 P_Get_Token_Seq .................. 29
2.3.21 P_Id ................................. 30
2.3.22 P_IfLoop ............................ 31
2.3.23 P_Include ........................... 34
2.3.24 P_Line ............................... 34
2.3.25 P_Scs_Declare ...................... 35
2.3.26 P_Struct ............................. 36
2.3.27 P_Sys_Include ...................... 37
2.3.28 P_Switch ............................. 37
2.3.29 P_Type_Def ........................... 38
2.3.30 P_TypeSpec .......................... 39
2.3.31 P.Undef ............................ 41
2.3.32 P_User_Include ..................... 41
2.3.33 Reset_Line .......................... 41
LIST OF FIGURES and TABLES

FIGURE 1: Generalized Relationship Between Subprograms. . . . . 4

FIGURE 2: Generalized Flowchart for REFORMAT.C. . . . . . . . . 8

FIGURE 3: Generalized Flowchart for Parse . . . . . . . . . . . 11

TABLE 1: Character Token Types. . . . . . . . . . . . . . . . . . . . 48

TABLE 2: Token Types for Reserved Words . . . . . . . . . . . . . 60
CHAPTER 1
INTRODUCTION

1.1 BACKGROUND

Two of the main criteria for the success of a particular language are 1) the clarity of program syntax, i.e., readability, and 2) its cost of use, primarily the cost of program maintenance [Pratt 1984]. Two qualities of a good program are 1) minimization of maintenance costs, and 2) flexibility, the ease of changing, expanding, or upgrading a program [Yourdon 1975]. The use of typographic style can be used to achieve these.

Typographic style is defined as the set of style characteristics concerned with the formatting and commenting of source code [Oman 1990]. One such style is the book format paradigm. This model incorporates macro- and micro-typographic styles which deal with overall program formatting and statement formatting, respectively [Oman 1990].

The macro-typographic conventions used in this project include the creation of a title page, a table of contents, the source code, and appendices. The macro-typographic conventions reflected within the source code include the listing of each function on a different page, the name of the file containing the function at the top of the page, and its page number at the bottom.
The micro-typographic conventions implemented in the source code include printing the function name of a function definition in a boldface font, italicizing the function name of a function call, consistent spacing for declarations and within statements, and consistent indentation of statements embedded within other statements. Also, the alignment of a closing bracket with the beginning of its controlling statement aids the programmer in visually grouping related statements.

The appendices include a listing of the contents of user-defined header files included in the program, if any, and a cross-reference index. This index lists each function defined or called in the program, the file name containing the function, its page number in the reformatted listing, the functions it calls and is called from, and their respective page numbers.

Oman and Cook conducted four experiments to test the effect of book format style on maintenance, comprehension, and readability of programs. Pascal and C programs were used. Each experiment consisted of two groups; one group used a program reformatted to book format and the other group used the original source program. The programs were equivalent in that they contained the identical source code, just represented differently. The results of each experiment concluded that the programs written in book format were easier to understand and maintain than the unformatted version [Oman 1990]. Even inexperienced programmers using the
formatted version completed the assignments earlier and with greater accuracy than did veteran programmers using the unformatted version. This follows Yourdon’s suggestion to format program listings to make them more readable [Yourdon 1975]. He states, "Your program listing should be pleasing to the eye, so that the reader will find it easy - even inviting - to follow."

1.2 PROJECT OVERVIEW

A primary concern of reformatting is freedom from errors [Kernighan 1976]. A reformatting program eliminates the introduction of human errors by reformatting text without retyping it. The purpose of this project was, therefore, to design and implement a program reformatter that will accept C source code as input and generate an equivalent C program listing in book format style as output. The idea for the project was initiated by Dr. Patrick Michaud who is extremely interested in the results of the experiments mentioned above. Dr. Michaud will use the program to conduct his own tests and experiments. I intend to use the program in future employment involving the maintenance of C programs. It will be a tool to speed up the process of understanding another programmer’s program.

The reformatter project consists of six individual subprograms that are linked, compiled, and run together as a single program. Figure 1 is a structure chart representing a generalized view of the relationships between the subprograms. For example, the link
Figure 1
Generalized Relationship Between Subprograms
between REFORMAT.C and PARSER.C. is interpreted as: functions within REFORMAT.C primarily call functions from within itself and from within PARSER.C.

The REFORMAT.C subprogram governs the overall flow of the program; the PARSER.C subprogram, or syntactical analyzer, processes the input program statement by statement; the SCANNER.C subprogram, or lexical analyzer, processes the input program token by token; the LISTING.C subprogram provides the functions for bolding and italicizing and generates the reformatted output listing; the INDEX.C subprogram tracks, using linked lists, the functions called by a specific function and the functions that call it; and the PROGOUT.C subprogram generates the supplemental files, title page, table of contents, listing of header files and index, which are combined along with the reformatted listing into a single output file.

1.3 ENVIRONMENT

The reformatter was written in Turbo C 2.0 and implemented on an IBM AT compatible Swan 286/12 PC running the DOS 4.01 operating system. The output was printed on a Panasonic KX-P1180 printer.
CHAPTER 2
PROGRAM DESCRIPTION

2.1 PROCEDURE

The C reformatter program was organized using the project feature of Turbo C. This feature compiles and links all individual subprograms of the overall program that are listed in the REFORMAT.PRJ file. In the following discussion, "Cprogram" refers to the C program that is to be reformatted, whereas "program" refers to the reformatter program. Variables needed in more than one function of a subprogram and/or needed by more than one subprogram are declared globally. Otherwise, variables are declared locally. Variables used by more than one subprogram are listed in the header file for the subprogram in which it is declared. Each subprogram, in turn, uses the #include statement to gain access to the header files it needs. With the exception of two functions, variables are not passed between functions.

To execute the program, type "reformat" at the DOS prompt, followed by any number of Cprograms. For example,

    C:\> reformat file1 file2 ...

This program assumes that the Cprograms and header files are in the subdirectory from which the program is called. The first Cprogram listed, file1 in the example above, is the program name used on the title page.
The reformatter program, listed in Appendix A, was used as input to itself to produce the output listing in Appendix B. The page numbers in Appendix A were produced by printing the source code files from WordPerfect and are not part of the original source code. The page numbers in Appendix B were produced by the reformatter program. The following discussions of the individual modules and the functions within them are in the same order as in the output listing. Refer to the table of contents of the output for the page number of the function listing. Also in the following discussions, filenames are printed in all uppercase letters, function names are printed in boldface, and variable names are underlined to distinguish them from the surrounding text.

2.2 REFORMAT.C

2.2.1 Main

Main controls the basic flow of the entire program as illustrated in Figure 2: an input file is processed until the end of file is reached. This process is repeated for each file listed on the command line.

Main first checks that a valid number of arguments have been entered on the command line, i.e., at least one file has been listed for reformatting. If not, a message is printed to the screen indicating the proper syntax to run the program, and the program terminates. If at least one program is listed, the
Figure 2
Generalized Flowchart For REFORMAT.C
program proceeds. After initializing a few variables, **Build_Reserve_Table** is called to create a table of reserved words. The file that will contain the reformatted C program, LISTING.PRG, is opened for write access and a pointer is assigned to it. The program name to be listed on the title page is passed to **Format_Progname** to be converted to all capital letters.

Beginning with the first file listed on the command line, and repeating for each file, the following steps are performed:

1. The filename is passed to **Format_Progname**.
2. The file is opened with read access and assigned to a pointer. If the file does not exist, an error message is printed to the screen.
3. An information message regarding the file currently being reformatted is printed to the screen.
4. The first character of the input file is read.
5. While not end of file, **Parse** is called.
6. If the last token of an input file is a comment, the page number is printed to advance to the next page.
6. The input file is closed.

**Main** then prints an information message indicating that the output is being prepared and calls **Output_Program** to generate the output, BOOK.TXT.
2.2.2 Build_Reserved_Table

Build_Reserved_Table creates a table of 39 entries consisting of the 32 keywords listed on page 192 of Kernighan and Ritchie's "The C Programming Language, 2nd Edition", the six preprocessors keywords listed on page 239 of the same book, and the FILE type name.

2.3 PARSER.C

PARSER.C consists of 37 functions. Of these functions, about half are called from the first and primary function of this module, Parse. PARSER.C is not a conventional parser in that function calls to proceed with the processing of a statement do not exactly follow the C grammar defined by Kernighan and Richie. The program assumes that the input program is grammatically correct, and "parses" to achieve the proper formatting. Figure 3 is a generalized flow chart representing a broad overview of the processing controlled by Parse.

2.3.1 Parse

Parse is a large switch statement directing the scanning based on the first token of a statement. In most cases, control does not return to Parse until the beginning of the next statement. Prior to the switch statement, several variables are initialized and carry_over is checked to determine whether or not a call to Scan is needed. A value of one indicates that the parsing of the previous statement included a token of the next statement to be
Figure 3
Generalized Flowchart For Parse