The c programming language

The C programming Language By Brian W. Kernighan and Dennis M. Ritchie. Published by Prentice-Hall in 1988 ISBN 0-13-110362-8 (paperback) ISBN 0-13-110370-9 Contents Preface Preface to the first edition Introduction 1. Chapter 1: A Tutorial Introduction 1. Getting Started 2. Variables and Arithmetic Expressions 3. The for statement 4. Symbolic Constants 5. Character Input and Output 1. File Copying 2. Character Counting 3. Line Counting 4. Word Counting 6. Arrays 7. Functions 8. Arguments - Call by Value 9. Character Arrays 10. External Variables and Scope 2. Chapter 2: Types, Operators and Expressions 1. Variable Names 2. Data Types and Sizes 3. Constants 4. Declarations 5. Arithmetic Operators 6. Relational and Logical Operators 7. Type Conversions 8. Increment and Decrement Operators 9. Bitwise Operators 10. Assignment Operators and Expressions 11. Conditional Expressions 12. Precedence and Order of Evaluation 3. Chapter 3: Control Flow 1. Statements and Blocks 2. If-Else 3. Else-If 4. Switch 5. Loops - While and For 6. Loops - Do-While 7. Break and Continue 8. Goto and labels 4. Chapter 4: Functions and Program Structure 1. Basics of Functions 2. Functions Returning Non-integers 3. External Variables 4. Scope Rules 5. Header Files 6. Static Variables 7. Register Variables 8. Block Structure 9. Initialization 10. Recursion 11. The C Preprocessor 1. File Inclusion 2. Macro Substitution 3. Conditional Inclusion 5. Chapter 5: Pointers and Arrays 1. Pointers and Addresses 2. Pointers and Function Arguments 3. Pointers and Arrays 4. Address Arithmetic 5. Character Pointers and Functions 6. Pointer Arrays; Pointers to Pointers 7. Multi-dimensional Arrays 8. Initialization of Pointer Arrays 9. Pointers vs. Multi-dimensional Arrays 10. Command-line Arguments 11. Pointers to Functions 12. Complicated Declarations 6. Chapter 6: Structures 1. Basics of Structures 2. Structures and Functions 3. Arrays of

Structures 4. Pointers to Structures 5. Self-referential Structures 6. Table Lookup 7. Typedef 8. Unions 9. Bit-fields 7. Chapter 7: Input and Output 1. Standard Input and Output 2. Formatted Output - printf 3. Variable-length Argument Lists 4. Formatted Input - Scanf 5. File Access 6. Error Handling - Stderr and Exit 7. Line Input and Output 8. Miscellaneous Functions 1. String Operations 2. Character Class Testing and Conversion 3. Ungetc 4. Command Execution 5. Storage Management 6. Mathematical Functions 7. Random Number generation 8. Chapter 8: The UNIX System Interface 1. File Descriptors 2. Low Level I/O - Read and Write 3. Open, Creat, Close, Unlink 4. Random Access - Lseek 5. Example - An implementation of Fopen and Getc 6. Example - Listing Directories 7. Example - A Storage Allocator Appendix A: Reference Manual 1. Introduction 2. Lexical Conventions 3. Syntax Notation 4. Meaning of Identifiers 5. Objects and Lvalues 6. Conversions 7. Expressions 8. Declarations 9. Statements 10. External Declarations 11. Scope and Linkage 12. Preprocessor 13. Grammar Appendix B: Standard Library 1. Input and Output: 1. File Operations 2. Formatted Output 3. Formatted Input 4. Character Input and Output Functions 5. Direct Input and Output Functions 6. File Positioning Functions 7. Error Functions 2. Character Class Tests: 3. String Functions: 4. Mathematical Functions: 5. Utility Functions: 6. Diagnostics: 7. Variable Argument Lists: 8. Non-local Jumps: 9. Signals: 10. Date and Time Functions: 11. Implementation-defined Limits: and Appendix C: Summary of Changes Preface The computing world has undergone a revolution since the publication of The C Programming Language in 1978. Big computers are much bigger, and personal computers have capabilities that rival mainframes of a decade ago. During this time, C has changed too, although only modestly, and it has spread far beyond its

origins as the language of the UNIX operating system. The growing popularity of C, the changes in the language over the years, and the creation of compilers by groups not involved in its design, combined to demonstrate a need for a more precise and more contemporary definition of the language than the first edition of this book provided. In 1983, the American National Standards Institute (ANSI) established a committee whose goal was to produce ‘‘ an unambiguous and machine-independent definition of the language C’’, while still retaining its spirit. The result is the ANSI standard for C. The standard formalizes constructions that were hinted but not described in the first edition, particularly structure assignment and enumerations. It provides a new form of function declaration that permits cross-checking of definition with use. It specifies a standard library, with an extensive set of functions for performing input and output, memory management, string manipulation, and similar tasks. It makes precise the behavior of features that were not spelled out in the original definition, and at the same time states explicitly which aspects of the language remain machine-dependent. This Second Edition of The C Programming Language describes C as defined by the ANSI standard. Although we have noted the places where the language has evolved, we have chosen to write exclusively in the new form. For the most part, this makes no significant difference; the most visible change is the new form of function declaration and definition. Modern compilers already support most features of the standard. We have tried to retain the brevity of the first edition. C is not a big language, and it is not well served by a big book. We have improved the exposition of critical features, such as pointers, that are central to C programming. We have refined the original examples, and have added new examples in several

chapters. For instance, the treatment of complicated declarations is augmented by programs that convert declarations into words and vice versa. As before, all examples have been tested directly from the text, which is in machine-readable form. Appendix A, the reference manual, is not the standard, but our attempt to convey the essentials of the standard in a smaller space. It is meant for easy comprehension by programmers, but not as a definition for compiler writers -- that role properly belongs to the standard itself. Appendix B is a summary of the facilities of the standard library. It too is meant for reference by programmers, not implementers. Appendix C is a concise summary of the changes from the original version. As we said in the preface to the first edition, C ‘‘ wears well as one’s experience with it grows’’. With a decade more experience, we still feel that way. We hope that this book will help you learn C and use it well. We are deeply indebted to friends who helped us to produce this second edition. Jon Bently, Doug Gwyn, Doug McIlroy, Peter Nelson, and Rob Pike gave us perceptive comments on almost every page of draft manuscripts. We are grateful for careful reading by Al Aho, Dennis Allison, Joe Campbell, G. R. Emlin, Karen Fortgang, Allen Holub, Andrew Hume, Dave Kristol, John Linderman, Dave Prosser, Gene Spafford, and Chris van Wyk. We also received helpful suggestions from Bill Cheswick, Mark Kernighan, Andy Koenig, Robin Lake, Tom London, Jim Reeds, Clovis Tondo, and Peter Weinberger. Dave Prosser answered many detailed questions about the ANSI standard. We used Bjarne Stroustrup’s C++ translator extensively for local testing of our programs, and Dave Kristol provided us with an ANSI C compiler for final testing. Rich Drechsler helped greatly with typesetting. Our sincere thanks to all. Brian W. Kernighan Dennis M. Ritchie Preface to the

first edition C is a general-purpose programming language with features economy of expression, modern flow control and data structures, and a rich set of operators. C is not a ‘‘ very high level’’ language, nor a ‘‘ big’’ one, and is not specialized to any particular area of application. But its absence of restrictions and its generality make it more convenient and effective for many tasks than supposedly more powerful languages. C was originally designed for and implemented on the UNIX operating system on the DEC PDP-11, by Dennis Ritchie. The operating system, the C compiler, and essentially all UNIX applications programs (including all of the software used to prepare this book) are written in C. Production compilers also exist for several other machines, including the IBM System/370, the Honeywell 6000, and the Interdata 8/32. C is not tied to any particular hardware or system, however, and it is easy to write programs that will run without change on any machine that supports C. This book is meant to help the reader learn how to program in C. It contains a tutorial introduction to get new users started as soon as possible, separate chapters on each major feature, and a reference manual. Most of the treatment is based on reading, writing and revising examples, rather than on mere statements of rules. For the most part, the examples are complete, real programs rather than isolated fragments. All examples have been tested directly from the text, which is in machine-readable form. Besides showing how to make effective use of the language, we have also tried where possible to illustrate useful algorithms and principles of good style and sound design. The book is not an introductory programming manual; it assumes some familiarity with basic programming concepts like variables, assignment statements, loops, and functions. Nonetheless, a novice programmer should be able to read along and pick up

the language, although access to more knowledgeable colleague will help. In our experience, C has proven to be a pleasant, expressive and versatile language for a wide variety of programs. It is easy to learn, and it wears well as on’s experience with it grows. We hope that this book will help you to use it well. The thoughtful criticisms and suggestions of many friends and colleagues have added greatly to this book and to our pleasure in writing it. In particular, Mike Bianchi, Jim Blue, Stu Feldman, Doug McIlroy Bill Roome, Bob Rosin and Larry Rosler all read multiple volumes with care. We are also indebted to Al Aho, Steve Bourne, Dan Dvorak, Chuck Haley, Debbie Haley, Marion Harris, Rick Holt, Steve Johnson, John Mashey, Bob Mitze, Ralph Muha, Peter Nelson, Elliot Pinson, Bill Plauger, Jerry Spivack, Ken Thompson, and Peter Weinberger for helpful comments at various stages, and to Mile Lesk and Joe Ossanna for invaluable assistance with typesetting. Brian W. Kernighan Dennis M. Ritchie Introduction C is a general-purpose programming language. It has been closely associated with the UNIX operating system where it was developed, since both the system and most of the programs that run on it are written in C. The language, however, is not tied to any one operating system or machine; and although it has been called a ‘‘ system programming language’’ because it is useful for writing compilers and operating systems, it has been used equally well to write major programs in many different domains. Many of the important ideas of C stem from the language BCPL, developed by Martin Richards. The influence of BCPL on C proceeded indirectly through the language B, which was written by Ken Thompson in 1970 for the first UNIX system on the DEC PDP-7. BCPL and B are ‘‘ typeless’’ languages. By contrast, C provides a variety of data types. The fundamental types are characters, and integers and floating point

numbers of several sizes. In addition, there is a hierarchy of derived data types created with pointers, arrays, structures and unions. Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement. Pointers provide for machine-independent address arithmetic. C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (if-else), selecting one of a set of possible values (switch), looping with the termination test at the top (while, for) or at the bottom (do), and early loop exit (break). Functions may return values of basic types, structures, unions, or pointers. Any function may be called recursively. Local variables are typically ‘‘ automatic’’, or created anew with each invocation. Function definitions may not be nested but variables may be declared in a block-structured fashion. The functions of a C program may exist in separate source files that are compiled separately. Variables may be internal to a function, external but known only within a single source file, or visible to the entire program. A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation. C is a relatively ‘‘ low-level’’ language. This characterization is not pejorative; it simply means that C deals with the same sort of objects that most computers do, namely characters, numbers, and addresses. These may be combined and moved about with the arithmetic and logical operators implemented by real machines. C provides no operations to deal directly with composite objects such as character strings, sets, lists or arrays. There are no operations that manipulate an entire array or string, although structures may be copied as a unit. The language does not define any storage allocation facility other than static definition and the stack discipline

provided by the local variables of functions; there is no heap or garbage collection. Finally, C itself provides no input/output facilities; there are no READ or WRITE statements, and no built-in file access methods. All of these higher-level mechanisms must be provided by explicitly called functions. Most C implementations have included a reasonably standard collection of such functions. Similarly, C offers only straightforward, single-thread control flow: tests, loops, grouping, and subprograms, but not multiprogramming, parallel operations, synchronization, or coroutines. Although the absence of some of these features may seem like a grave deficiency, (‘‘ You mean I have to call a function to compare two character strings?’’), keeping the language down to modest size has real benefits. Since C is relatively small, it can be described in small space, and learned quickly. A programmer can reasonably expect to know and understand and indeed regularly use the entire language. For many years, the definition of C was the reference manual in the first edition of The C Programming Language. In 1983, the American National Standards Institute (ANSI) established a committee to provide a modern, comprehensive definition of C. The resulting definition, the ANSI standard, or ‘‘ ANSI C’’, was completed in late 1988. Most of the features of the standard are already supported by modern compilers. The standard is based on the original reference manual. The language is relatively little changed; one of the goals of the standard was to make sure that most existing programs would remain valid, or, failing that, that compilers could produce warnings of new behavior. For most programmers, the most important change is the new syntax for declaring and defining functions. A function declaration can now include a description of the arguments of the function; the definition syntax changes to match. This

extra information makes it much easier for compilers to detect errors caused by mismatched arguments; in our experience, it is a very useful addition to the language. There are other small-scale language changes. Structure assignment and enumerations, which had been widely available, are now officially part of the language. Floating-point computations may now be done in single precision. The properties of arithmetic, especially for unsigned types, are clarified. The preprocessor is more elaborate. Most of these changes will have only minor effects on most programmers. A second significant contribution of the standard is the definition of a library to accompany C. It specifies functions for accessing the operating system (for instance, to read and write files), formatted input and output, memory allocation, string manipulation, and the like. A collection of standard headers provides uniform access to declarations of functions in data types. Programs that use this library to interact with a host system are assured of compatible behavior. Most of the library is closely modeled on the ‘‘ standard I/O library’’ of the UNIX system. This library was described in the first edition, and has been widely used on other systems as well. Again, most programmers will not see much change. Because the data types and control structures provided by C are supported directly by most computers, the run-time library required to implement self-contained programs is tiny. The standard library functions are only called explicitly, so they can be avoided if they are not needed. Most can be written in C, and except for the operating system details they conceal, are themselves portable. Although C matches the capabilities of many computers, it is independent of any particular machine architecture. With a little care it is easy to write portable programs, that is, programs that can be run without change on a variety of hardware. The

standard makes portability issues explicit, and prescribes a set of constants that characterize the machine on which the program is run. C is not a strongly-typed language, but as it has evolved, its type-checking has been strengthened. The original definition of C frowned on, but permitted, the interchange of pointers and integers; this has long since been eliminated, and the standard now requires the proper declarations and explicit conversions that had already been enforced by good compilers. The new function declarations are another step in this direction. Compilers will warn of most type errors, and there is no automatic conversion of incompatible data types. Nevertheless, C retains the basic philosophy that programmers know what they are doing; it only requires that they state their intentions explicitly. C, like any other language, has its blemishes. Some of the operators have the wrong precedence; some parts of the syntax could be better. Nonetheless, C has proven to ben an extremely effective and expressive language for a wide variety of programming applications. The book is organized as follows. Chapter 1 is a tutorial on the central part of C. The purpose is to get the reader started as quickly as possible, since we believe strongly that the way to learn a new language is to write programs in it. The tutorial does assume a working knowledge of the basic elements of programming; there is no explanation of computers, of compilation, nor of the meaning of an expression like n= n+1. Although we have tried where possible to show useful programming techniques, the book is not intended to be a reference work on data structures and algorithms; when forced to make a choice, we have concentrated on the language. Chapters 2 through 6 discuss various aspects of C in more detail, and rather more formally, than does Chapter 1, although the emphasis is still on examples of complete

programs, rather than isolated fragments. Chapter 2 deals with the basic data types, operators and expressions. Chapter 3 threats control flow: if-else, switch, while, for, etc. Chapter 4 covers functions and program structure - external variables, scope rules, multiple source files, and so on - and also touches on the preprocessor. Chapter 5 discusses pointers and address arithmetic. Chapter 6 covers structures and unions. Chapter 7 describes the standard library, which provides a common interface to the operating system. This library is defined by the ANSI standard and is meant to be supported on all machines that support C, so programs that use it for input, output, and other operating system access can be moved from one system to another without change. Chapter 8 describes an interface between C programs and the UNIX operating system, concentrating on input/output, the file system, and storage allocation. Although some of this chapter is specific to UNIX systems, programmers who use other systems should still find useful material here, including some insight into how one version of the standard library is implemented, and suggestions on portability. Appendix A contains a language reference manual. The official statement of the syntax and semantics of the C language is the ANSI standard itself. That document, however, is intended foremost for compiler writers. The reference manual here conveys the definition of the language more concisely and without the same legalistic style. Appendix B is a summary of the standard library, again for users rather than implementers. Appendix C is a short summary of changes from the original language. In cases of doubt, however, the standard and one’s own compiler remain the final authorities on the language. Chapter 1 - A Tutorial Introduction Let us begin with a quick introduction in C. Our aim is to show the essential elements of the language

in real programs, but without getting bogged down in details, rules, and exceptions. At this point, we are not trying to be complete or even precise (save that the examples are meant to be correct). We want to get you as quickly as possible to the point where you can write useful programs, and to do that we have to concentrate on the basics: variables and constants, arithmetic, control flow, functions, and the rudiments of input and output. We are intentionally leaving out of this chapter features of C that are important for writing bigger programs. These include pointers, structures, most of C’s rich set of operators, several control-flow statements, and the standard library. This approach and its drawbacks. Most notable is that the complete story on any particular feature is not found here, and the tutorial, by being brief, may also be misleading. And because the examples do not use the full power of C, they are not as concise and elegant as they might be. We have tried to minimize these effects, but be warned. Another drawback is that later chapters will necessarily repeat some of this chapter. We hope that the repetition will help you more than it annoys. In any case, experienced programmers should be able to extrapolate from the material in this chapter to their own programming needs. Beginners should supplement it by writing small, similar programs of their own. Both groups can use it as a framework on which to hang the more detailed descriptions that begin in Chapter 2. 1. 1 Getting Started The only way to learn a new programming language is by writing programs in it. The first program to write is the same for all languages: Print the words hello, world This is a big hurdle; to leap over it you have to be able to create the program text somewhere, compile it successfully, load it, run it, and find out where your output went. With these mechanical details mastered, everything else is comparatively easy. In C, the

program to print ‘‘ hello, world’’ is #include main() { printf(" hello, world"); } Just how to run this program depends on the system you are using. As a specific example, on the UNIX operating system you must create the program in a file whose name ends in ‘‘. c’’, such as hello. c, then compile it with the command cc hello. c If you haven’t botched anything, such as omitting a character or misspelling something, the compilation will proceed silently, and make an executable file called a. out. If you run a. out by typing the command a. out it will print hello, world On other systems, the rules will be different; check with a local expert. Now, for some explanations about the program itself. A C program, whatever its size, consists of functions and variables. A function contains statements that specify the computing operations to be done, and variables store values used during the computation. C functions are like the subroutines and functions in Fortran or the procedures and functions of Pascal. Our example is a function named main. Normally you are at liberty to give functions whatever names you like, but ‘‘ main’’ is special - your program begins executing at the beginning of main. This means that every program must have a main somewhere. will usually call other functions to help perform its job, some that you wrote, and others from libraries that are provided for you. The first line of the program, main #include tells the compiler to include information about the standard input/output library; the line appears at the beginning of many C source files. The standard library is described in Chapter 7 and Appendix B. One method of communicating data between functions is for the calling function to provide a list of values, called arguments, to the function it calls. The parentheses after the function name surround the argument list. In this example, main is defined to be a function that expects no arguments, which

is indicated by the empty list ( ). #include main() { printf(" hello, world"); } include information about standard library define a function called main that received no argument values statements of main are enclosed in braces main calls library function printf to print this sequence of characters represents the newline character The first C program The statements of a function are enclosed in braces { }. The function main contains only one statement, printf(" hello, world"); A function is called by naming it, followed by a parenthesized list of arguments, so this calls the function printf with the argument " hello, world". printf is a library function that prints output, in this case the string of characters between the quotes. A sequence of characters in double quotes, like " hello, world", is called a character string or string constant. For the moment our only use of character strings will be as arguments for printf and other functions. The sequence in the string is C notation for the newline character, which when printed advances the output to the left margin on the next line. If you leave out the (a worthwhile experiment), you will find that there is no line advance after the output is printed. You must use to include a newline character in the printf argument; if you try something like printf(" hello, world "); the C compiler will produce an error message. never supplies a newline character automatically, so several calls may be used to build up an output line in stages. Our first program could just as well have been written printf #include main() { printf(" hello, "); printf(" world"); printf(""); } to produce identical output. Notice that represents only a single character. An escape sequence like provides a general and extensible mechanism for representing hard-to-type or invisible characters. Among the others that C provides are for tab, for backspace, " for the double quote and \\ for the backslash itself. There is a complete list in

Section 2. 3. Exercise 1-1. Run the ‘‘ hello, world’’ program on your system. Experiment with leaving out parts of the program, to see what error messages you get. Exercise 1-2. Experiment to find out what happens when prints’s argument string contains c, where c is some character not listed above. 1. 2 Variables and Arithmetic Expressions The next program uses the formula oC=(5/9)(oF-32) to print the following table of Fahrenheit temperatures and their centigrade or Celsius equivalents: 1 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 -17 -6 4 15 26 37 48 60 71 82 93 104 115 126 137 148 The program itself still consists of the definition of a single function named main. It is longer than the one that printed ‘‘ hello, world’’, but not complicated. It introduces several new ideas, including comments, declarations, variables, arithmetic expressions, loops , and formatted output. #include /* print Fahrenheit-Celsius table for fahr = 0, 20, ..., 300 */ main() { int fahr, celsius; int lower, upper, step; lower = 0; upper = 300; step = 20; /* lower limit of temperature scale */ /* upper limit */ /* step size */ fahr = lower; while (fahr max) { max = len; copy(); } if (max > 0) /* there was a line */ printf("%s", longest); return 0; } /* getline: specialized version */ int getline(void) { int c, i; extern char line[]; for (i = 0; i < MAXLINE - 1 && (c= getchar)) != EOF && c != ’’; ++i) line[i] = c; if (c == ’’) { line[i] = c; ++i; } line[i] = ’ 0’; return i; } /* copy: specialized version */ void copy(void) { int i; extern char line[], longest[]; i = 0; while ((longest[i] = line[i]) != ’ 0’) ++i; } /* maximum input line size */ /* maximum length seen so far */ /* current input line */ /* longest line saved here */ The external variables in main, getline and copy are defined by the first lines of the example above, which state their type and cause storage to be allocated for them. Syntactically, external definitions are just like definitions of local

variables, but since they occur outside of functions, the variables are external. Before a function can use an external variable, the name of the variable must be made known to the function; the declaration is the same as before except for the added keyword extern. In certain circumstances, the extern declaration can be omitted. If the definition of the external variable occurs in the source file before its use in a particular function, then there is no need for an extern declaration in the function. The extern declarations in main, getline and copy are thus redundant. In fact, common practice is to place definitions of all external variables at the beginning of the source file, and then omit all extern declarations. If the program is in several source files, and a variable is defined in file1 and used in file2 and file3, then extern declarations are needed in file2 and file3 to connect the occurrences of the variable. The usual practice is to collect extern declarations of variables and functions in a separate file, historically called a header, that is included by #include at the front of each source file. The suffix . h is conventional for header names. The functions of the standard library, for example, are declared in headers like . This topic is discussed at length in Chapter 4, and the library itself in Chapter 7 and Appendix B. Since the specialized versions of getline and copy have no arguments, logic would suggest that their prototypes at the beginning of the file should be getline() and copy(). But for compatibility with older C programs the standard takes an empty list as an old-style declaration, and turns off all argument list checking; the word void must be used for an explicitly empty list. We will discuss this further in Chapter 4. You should note that we are using the words definition and declaration carefully when we refer to external variables in this section.‘‘ Definition’’ refers to the place where the variable is created or assigned

storage; ‘‘ declaration’’ refers to places where the nature of the variable is stated but no storage is allocated. By the way, there is a tendency to make everything in sight an extern variable because it appears to simplify communications - argument lists are short and variables are always there when you want them. But external variables are always there even when you don’t want them. Relying too heavily on external variables is fraught with peril since it leads to programs whose data connections are not all obvious - variables can be changed in unexpected and even inadvertent ways, and the program is hard to modify. The second version of the longest-line program is inferior to the first, partly for these reasons, and partly because it destroys the generality of two useful functions by writing into them the names of the variables they manipulate. At this point we have covered what might be called the conventional core of C. With this handful of building blocks, it’s possible to write useful programs of considerable size, and it would probably be a good idea if you paused long enough to do so. These exercises suggest programs of somewhat greater complexity than the ones earlier in this chapter. Exercise 1-20. Write a program detab that replaces tabs in the input with the proper number of blanks to space to the next tab stop. Assume a fixed set of tab stops, say every n columns. Should n be a variable or a symbolic parameter? Exercise 1-21. Write a program entab that replaces strings of blanks by the minimum number of tabs and blanks to achieve the same spacing. Use the same tab stops as for detab. When either a tab or a single blank would suffice to reach a tab stop, which should be given preference? Exercise 1-22. Write a program to ‘‘ fold’’ long input lines into two or more shorter lines after the last non-blank character that occurs before the n-th column of input. Make sure your program does something

intelligent with very long lines, and if there are no blanks or tabs before the specified column. Exercise 1-23. Write a program to remove all comments from a C program. Don’t forget to handle quoted strings and character constants properly. C comments don’t nest. Exercise 1-24. Write a program to check a C program for rudimentary syntax errors like unmatched parentheses, brackets and braces. Don’t forget about quotes, both single and double, escape sequences, and comments. (This program is hard if you do it in full generality.) Chapter 2 - Types, Operators and Expressions Variables and constants are the basic data objects manipulated in a program. Declarations list the variables to be used, and state what type they have and perhaps what their initial values are. Operators specify what is to be done to them. Expressions combine variables and constants to produce new values. The type of an object determines the set of values it can have and what operations can be performed on it. These building blocks are the topics of this chapter. The ANSI standard has made many small changes and additions to basic types and expressions. There are now signed and unsigned forms of all integer types, and notations for unsigned constants and hexadecimal character constants. Floating-point operations may be done in single precision; there is also a long double type for extended precision. String constants may be concatenated at compile time. Enumerations have become part of the language, formalizing a feature of long standing. Objects may be declared const, which prevents them from being changed. The rules for automatic coercions among arithmetic types have been augmented to handle the richer set of types. 2. 1 Variable Names Although we didn’t say so in Chapter 1, there are some restrictions on the names of variables and symbolic constants. Names are made up of letters and digits; the first

character must be a letter. The underscore ‘‘ _’’ counts as a letter; it is sometimes useful for improving the readability of long variable names. Don’t begin variable names with underscore, however, since library routines often use such names. Upper and lower case letters are distinct, so x and X are two different names. Traditional C practice is to use lower case for variable names, and all upper case for symbolic constants. At least the first 31 characters of an internal name are significant. For function names and external variables, the number may be less than 31, because external names may be used by assemblers and loaders over which the language has no control. For external names, the standard guarantees uniqueness only for 6 characters and a single case. Keywords like if, else, int, float, etc., are reserved: you can’t use them as variable names. They must be in lower case. It’s wise to choose variable names that are related to the purpose of the variable, and that are unlikely to get mixed up typographically. We tend to use short names for local variables, especially loop indices, and longer names for external variables. 2. 2 Data Types and Sizes There are only a few basic data types in C: a single byte, capable of holding one character in the local character set int an integer, typically reflecting the natural size of integers on the host machine float single-precision floating point double double-precision floating point char In addition, there are a number of qualifiers that can be applied to these basic types. short and long apply to integers: short int sh; long int counter; The word int can be omitted in such declarations, and typically it is. The intent is that short and long should provide different lengths of integers where practical; int will normally be the natural size for a particular machine. short is often 16 bits long, and int either 16 or 32 bits. Each compiler is free to choose appropriate sizes for its

own hardware, subject only to the the restriction that shorts and ints are at least 16 bits, longs are at least 32 bits, and short is no longer than int, which is no longer than long. The qualifier signed or unsigned may be applied to char or any integer. unsigned numbers are always positive or zero, and obey the laws of arithmetic modulo 2n, where n is the number of bits in the type. So, for instance, if chars are 8 bits, unsigned char variables have values between 0 and 255, while signed chars have values between -128 and 127 (in a two’s complement machine.) Whether plain chars are signed or unsigned is machine-dependent, but printable characters are always positive. The type long double specifies extended-precision floating point. As with integers, the sizes of floating-point objects are implementation-defined; float, double and long double could represent one, two or three distinct sizes. The standard headers and contain symbolic constants for all of these sizes, along with other properties of the machine and compiler. These are discussed in Appendix B. Exercise 2-1. Write a program to determine the ranges of char, short, int, and long variables, both signed and unsigned, by printing appropriate values from standard headers and by direct computation. Harder if you compute them: determine the ranges of the various floating-point types. 2. 3 Constants An integer constant like 1234 is an int. A long constant is written with a terminal l (ell) or L, as in 123456789L; an integer constant too big to fit into an int will also be taken as a long. Unsigned constants are written with a terminal u or U, and the suffix ul or UL indicates unsigned long. Floating-point constants contain a decimal point (123. 4) or an exponent (1e-2) or both; their type is double, unless suffixed. The suffixes f or F indicate a float constant; l or L indicate a long double. The value of an integer can be specified in octal or hexadecimal

instead of decimal. A leading 0 (zero) on an integer constant means octal; a leading 0x or 0X means hexadecimal. For example, decimal 31 can be written as 037 in octal and 0x1f or 0x1F in hex. Octal and hexadecimal constants may also be followed by L to make them long and U to make them unsigned: 0XFUL is an unsigned long constant with value 15 decimal. A character constant is an integer, written as one character within single quotes, such as ’x’. The value of a character constant is the numeric value of the character in the machine’s character set. For example, in the ASCII character set the character constant ’0’ has the value 48, which is unrelated to the numeric value 0. If we write ’0’ instead of a numeric value like 48 that depends on the character set, the program is independent of the particular value and easier to read. Character constants participate in numeric operations just as any other integers, although they are most often used in comparisons with other characters. Certain characters can be represented in character and string constants by escape sequences like (newline); these sequences look like two characters, but represent only one. In addition, an arbitrary byte-sized bit pattern can be specified by ’ooo’ where ooo is one to three octal digits (0... 7) or by ’xhh’ where hh is one or more hexadecimal digits (0... 9, a... f, A... F). So we might write #define VTAB ’ 013’ #define BELL ’ 007’ /* ASCII vertical tab */ /* ASCII bell character */ or, in hexadecimal, #define VTAB ’xb’ #define BELL ’x7’ /* ASCII vertical tab */ /* ASCII bell character */ The complete set of escape sequences is a f v alert (bell) character backspace formfeed newline carriage return horizontal tab vertical tab backslash ? question mark ’ single quote " double quote ooo octal number xhh hexadecimal number The character constant ’ 0’ represents the character with value zero, the null character. ’ 0’ is often

written instead of 0 to emphasize the character nature of some expression, but the numeric value is just 0. A constant expression is an expression that involves only constants. Such expressions may be evaluated at during compilation rather than run-time, and accordingly may be used in any place that a constant can occur, as in #define MAXLINE 1000 char line[MAXLINE+1]; or #define LEAP 1 /* in leap years */ int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31]; A string constant, or string literal, is a sequence of zero or more characters surrounded by double quotes, as in " I am a string" or "" /* the empty string */ The quotes are not part of the string, but serve only to delimit it. The same escape sequences used in character constants apply in strings; " represents the double-quote character. String constants can be concatenated at compile time: " hello, " " world" is equivalent to " hello, world" This is useful for splitting up long strings across several source lines. Technically, a string constant is an array of characters. The internal representation of a string has a null character ’ 0’ at the end, so the physical storage required is one more than the number of characters written between the quotes. This representation means that there is no limit to how long a string can be, but programs must scan a string completely to determine its length. The standard library function strlen(s) returns the length of its character string argument s, excluding the terminal ’ 0’. Here is our version: /* strlen: return length of s */ int strlen(char s[]) { int i; while (s[i] != ’ 0’) ++i; return i; } strlen and other string functions are declared in the standard header . Be careful to distinguish between a character constant and a string that contains a single character: ’x’ is not the same as " x". The former is an integer, used to produce the numeric value of the letter x in the machine’s

character set. The latter is an array of characters that contains one character (the letter x) and a ’ 0’. There is one other kind of constant, the enumeration constant. An enumeration is a list of constant integer values, as in enum boolean { NO, YES }; The first name in an enum has value 0, the next 1, and so on, unless explicit values are specified. If not all values are specified, unspecified values continue the progression from the last specified value, as the second of these examples: enum escapes { BELL = ’a’, BACKSPACE = ’’, TAB = ’’, NEWLINE = ’’, VTAB = ’v’, RETURN = ’ ’ }; enum months { JAN = 1, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV, DEC }; /* FEB = 2, MAR = 3, etc. */ Names in different enumerations must be distinct. Values need not be distinct in the same enumeration. Enumerations provide a convenient way to associate constant values with names, an alternative to #define with the advantage that the values can be generated for you. Although variables of enum types may be declared, compilers need not check that what you store in such a variable is a valid value for the enumeration. Nevertheless, enumeration variables offer the chance of checking and so are often better than #defines. In addition, a debugger may be able to print values of enumeration variables in their symbolic form. 2. 4 Declarations All variables must be declared before use, although certain declarations can be made implicitly by content. A declaration specifies a type, and contains a list of one or more variables of that type, as in int lower, upper, step; char c, line[1000]; Variables can be distributed among declarations in any fashion; the lists above could well be written as int int int char char lower; upper; step; c; line[1000]; The latter form takes more space, but is convenient for adding a comment to each declaration for subsequent modifications. A variable may also be initialized in its declaration. If the name is followed by an equals sign

and an expression, the expression serves as an initializer, as in char int int float esc = '\\'; i = 0; limit = MAXLINE+1; eps = 1. 0e-5; If the variable in question is not automatic, the initialization is done once only, conceptionally before the program starts executing, and the initializer must be a constant expression. An explicitly initialized automatic variable is initialized each time the function or block it is in is entered; the initializer may be any expression. External and static variables are initialized to zero by default. Automatic variables for which is no explicit initializer have undefined (i. e., garbage) values. The qualifier const can be applied to the declaration of any variable to specify that its value will not be changed. For an array, the const qualifier says that the elements will not be altered. const double e = 2. 71828182845905; const char msg[] = " warning: "; The const declaration can also be used with array arguments, to indicate that the function does not change that array: int strlen(const char[]); The result is implementation-defined if an attempt is made to change a const. 2. 5 Arithmetic Operators The binary arithmetic operators are +, -, *, /, and the modulus operator %. Integer division truncates any fractional part. The expression x % y produces the remainder when x is divided by y, and thus is zero when y divides x exactly. For example, a year is a leap year if it is divisible by 4 but not by 100, except that years divisible by 400 are leap years. Therefore if ((year % 4 == 0 && year % 100 != 0) || year % 400 == 0) printf("%d is a leap year", year); else printf("%d is not a leap year", year); The % operator cannot be applied to a float or double. The direction of truncation for / and the sign of the result for % are machine-dependent for negative operands, as is the action taken on overflow or underflow. The binary + and - operators have the same precedence, which is lower than the precedence of *, / and %, which is

in turn lower than unary + and -. Arithmetic operators associate left to right. Table 2. 1 at the end of this chapter summarizes precedence and associativity for all operators. 2. 6 Relational and Logical Operators The relational operators are > >= < = ’0’ && s[i] = ’A’ && c = ’0’ && c j and logical expressions connected by && and || are defined to have value 1 if true, and 0 if false. Thus the assignment d = c >= ’0’ && c 1UL because -1L is promoted to unsigned long and thus appears to be a large positive number. Conversions take place across assignments; the value of the right side is converted to the type of the left, which is the type of the result. A character is converted to an integer, either by sign extension or not, as described above. Longer integers are converted to shorter ones or to chars by dropping the excess high-order bits. Thus in int i; char c; i = c; c = i; the value of c is unchanged. This is true whether or not sign extension is involved. Reversing the order of assignments might lose information, however. If x is float and i is int, then x = i and i = x both cause conversions; float to int causes truncation of any fractional part. When a double is converted to float, whether the value is rounded or truncated is implementation dependent. Since an argument of a function call is an expression, type conversion also takes place when arguments are passed to functions. In the absence of a function prototype, char and short become int, and float becomes double. This is why we have declared function arguments to be int and double even when the function is called with char and float. Finally, explicit type conversions can be forced (‘‘ coerced’’) in any expression, with a unary operator called a cast. In the construction (type name) expression the expression is converted to the named type by the conversion rules above. The precise meaning of a cast is as if the expression

were assigned to a variable of the specified type, which is then used in place of the whole construction. For example, the library routine sqrt expects a double argument, and will produce nonsense if inadvertently handled something else. (sqrt is declared in .) So if n is an integer, we can use sqrt((double) n) to convert the value of n to double before passing it to sqrt. Note that the cast produces the value of n in the proper type; n itself is not altered. The cast operator has the same high precedence as other unary operators, as summarized in the table at the end of this chapter. If arguments are declared by a function prototype, as the normally should be, the declaration causes automatic coercion of any arguments when the function is called. Thus, given a function prototype for sqrt: double sqrt(double) the call root2 = sqrt(2) coerces the integer 2 into the double value 2. 0 without any need for a cast. The standard library includes a portable implementation of a pseudo-random number generator and a function for initializing the seed; the former illustrates a cast: unsigned long int next = 1; /* rand: return pseudo-random integer on 0.. 32767 */ int rand(void) { next = next * 1103515245 + 12345; return (unsigned int) (next/65536) % 32768; } /* srand: set seed for rand() */ void srand(unsigned int seed) { next = seed; } Exercise 2-3. Write a function htoi(s), which converts a string of hexadecimal digits (including an optional 0x or 0X) into its equivalent integer value. The allowable digits are 0 through 9, a through f, and A through F. 2. 8 Increment and Decrement Operators C provides two unusual operators for incrementing and decrementing variables. The increment operator ++ adds 1 to its operand, while the decrement operator -- subtracts 1. We have frequently used ++ to increment variables, as in if (c == ’’) ++nl; The unusual aspect is that ++ and -- may be used either as

prefix operators (before the variable, as in ++n), or postfix operators (after the variable: n++). In both cases, the effect is to increment n. But the expression ++n increments n before its value is used, while n++ increments n after its value has been used. This means that in a context where the value is being used, not just the effect, ++n and n++ are different. If n is 5, then x = n++; sets x to 5, but x = ++n; sets x to 6. In both cases, n becomes 6. The increment and decrement operators can only be applied to variables; an expression like (i+j)++ is illegal. In a context where no value is wanted, just the incrementing effect, as in if (c == ’’) nl++; prefix and postfix are the same. But there are situations where one or the other is specifically called for. For instance, consider the function squeeze(s, c), which removes all occurrences of the character c from the string s. /* squeeze: delete all c from s */ void squeeze(char s[], int c) { int i, j; for (i = j = 0; s[i] != ’ 0’; i++) if (s[i] != c) s[j++] = s[i]; s[j] = ’ 0’; } Each time a non-c occurs, it is copied into the current j position, and only then is j incremented to be ready for the next character. This is exactly equivalent to if (s[i] != c) { s[j] = s[i]; j++; } Another example of a similar construction comes from the getline function that we wrote in Chapter 1, where we can replace if (c == ’’) { s[i] = c; + +i; } by the more compact if (c == ’’) s[i++] = c; As a third example, consider the standard function strcat(s, t), which concatenates the string t to the end of string s. strcat assumes that there is enough space in s to hold the combination. As we have written it, strcat returns no value; the standard library version returns a pointer to the resulting string. /* strcat: concatenate t to end of s; s must be big enough */ void strcat(char s[], char t[]) { int i, j; i = j = 0; while (s[i] != ’ 0’) /* find end of s */ i++; while ((s[i++] = t[j++]) != ’ 0’) /* copy t */ ; } As each member is copied from t to s, the postfix ++ is

applied to both i and j to make sure that they are in position for the next pass through the loop. Exercise 2-4. Write an alternative version of squeeze(s1, s2) that deletes each character in s1 that matches any character in the string s2. Exercise 2-5. Write the function any(s1, s2), which returns the first location in a string s1 where any character from the string s2 occurs, or -1 if s1 contains no characters from s2. (The standard library function strpbrk does the same job but returns a pointer to the location.) 2. 9 Bitwise Operators C provides six operators for bit manipulation; these may only be applied to integral operands, that is, char, short, int, and long, whether signed or unsigned. bitwise AND | bitwise inclusive OR ^ bitwise exclusive OR > right shift ~ one’s complement (unary) & The bitwise AND operator & is often used to mask off some set of bits, for example n = n & 0177; sets to zero all but the low-order 7 bits of n. The bitwise OR operator | is used to turn bits on: x = x | SET_ON; sets to one in x the bits that are set to one in SET_ON. The bitwise exclusive OR operator ^ sets a one in each bit position where its operands have different bits, and zero where they are the same. One must distinguish the bitwise operators & and | from the logical operators && and ||, which imply left-to-right evaluation of a truth value. For example, if x is 1 and y is 2, then x & y is zero while x && y is one. The shift operators > perform left and right shifts of their left operand by the number of bit positions given by the right operand, which must be non-negative. Thus x > (p+1-n)) & ~(~0 > (p+1-n) moves the desired field to the right end of the word. ~0 all 1-bits; shifting it left n positions is with ~0>= 1) if (x & 01) b++; return b; } Declaring the argument x to be an unsigned ensures that when it is right-shifted, vacated bits will be filled with zeros, not sign bits, regardless of the machine the program is run on. Quite apart from

conciseness, assignment operators have the advantage that they correspond better to the way people think. We say ‘‘ add 2 to i’’ or ‘‘ increment i by 2’’, not ‘‘ take i, add 2, then put the result back in i’’. Thus the expression i += 2 is preferable to i = i+2. In addition, for a complicated expression like yyval[yypv[p3+p4] + yypv[p1]] += 2 the assignment operator makes the code easier to understand, since the reader doesn’t have to check painstakingly that two long expressions are indeed the same, or to wonder why they’re not. And an assignment operator may even help a compiler to produce efficient code. We have already seen that the assignment statement has a value and can occur in expressions; the most common example is while ((c = getchar()) != EOF) ... The other assignment operators (+=, -=, etc.) can also occur in expressions, although this is less frequent. In all such expressions, the type of an assignment expression is the type of its left operand, and the value is the value after the assignment. Exercise 2-9. In a two’s complement number system, x &= (x-1) deletes the rightmost 1-bit in x. Explain why. Use this observation to write a faster version of bitcount. 2. 11 Conditional Expressions The statements if (a > b) z = a; else z = b; compute in z the maximum of a and b. The conditional expression, written with the ternary operator ‘‘?:’’, provides an alternate way to write this and similar constructions. In the expression expr1 ? expr2 : expr3 the expression expr1 is evaluated first. If it is non-zero (true), then the expression expr2 is evaluated, and that is the value of the conditional expression. Otherwise expr3 is evaluated, and that is the value. Only one of expr2 and expr3 is evaluated. Thus to set z to the maximum of a and b, z = (a > b) ? a : b; /* z = max(a, b) */ It should be noted that the conditional expression is indeed an expression, and it can be used wherever any other expression can be. If

expr2 and expr3 are of different types, the type of the result is determined by the conversion rules discussed earlier in this chapter. For example, if f is a float and n an int, then the expression (n > 0) ? f : n is of type float regardless of whether n is positive. Parentheses are not necessary around the first expression of a conditional expression, since the precedence of ?: is very low, just above assignment. They are advisable anyway, however, since they make the condition part of the expression easier to see. The conditional expression often leads to succinct code. For example, this loop prints n elements of an array, 10 per line, with each column separated by one blank, and with each line (including the last) terminated by a newline. for (i = 0; i < n; i++) printf("%6d%c", a[i], (i%10== 9 || i== n-1) ? ’’ : ’ ’); A newline is printed after every tenth element, and after the n-th. All other elements are followed by one blank. This might look tricky, but it’s more compact than the equivalent if-else. Another good example is printf(" You have %d items%s.", n, n== 1 ? "" : " s"); Exercise 2-10. Rewrite the function lower, which converts upper case letters to lower case, with a conditional expression instead of if-else. 2. 12 Precedence and Order of Evaluation Table 2. 1 summarizes the rules for precedence and associativity of all operators, including those that we have not yet discussed. Operators on the same line have the same precedence; rows are in order of decreasing precedence, so, for example, *, /, and % all have the same precedence, which is higher than that of binary + and -. The ‘‘ operator’’ () refers to function call. The operators -> and . are used to access members of structures; they will be covered in Chapter 6, along with sizeof (size of an object). Chapter 5 discusses * (indirection through a pointer) and & (address of an object), and Chapter 3 discusses the comma operator. Associativity () [] -> . left to right !

~ ++ -- + - * (type) sizeof right to left * / % left to right + left to right > left to right < >= left to right == != left to right & left to right ^ left to right | left to right && left to right || left to right ?: right to left = += -= *= /= %= &= ^= |= = right to left , left to right Unary & +, -, and * have higher precedence than the binary forms. Table 2. 1: Precedence and Associativity of Operators Operators Note that the precedence of the bitwise operators &, ^, and | falls below == and !=. This implies that bit-testing expressions like if ((x & MASK) == 0) ... must be fully parenthesized to give proper results. C, like most languages, does not specify the order in which the operands of an operator are evaluated. (The exceptions are &&, ||, ?:, and ‘,’.) For example, in a statement like x = f() + g(); f may be evaluated before g or vice versa; thus if either f or g alters a variable on which the other depends, x can depend on the order of evaluation. Intermediate results can be stored in temporary variables to ensure a particular sequence. Similarly, the order in which function arguments are evaluated is not specified, so the statement printf("%d %d", ++n, power(2, n)); /* WRONG */ can produce different results with different compilers, depending on whether n is incremented before power is called. The solution, of course, is to write ++n; printf("%d %d", n, power(2, n)); Function calls, nested assignment statements, and increment and decrement operators cause ‘‘ side effects’’ - some variable is changed as a by-product of the evaluation of an expression. In any expression involving side effects, there can be subtle dependencies on the order in which variables taking part in the expression are updated. One unhappy situation is typified by the statement a[i] = i++; The question is whether the subscript is the old value of i or the new. Compilers can interpret this in different ways, and generate different answers depending on

their interpretation. The standard intentionally leaves most such matters unspecified. When side effects (assignment to variables) take place within an expression is left to the discretion of the compiler, since the best order depends strongly on machine architecture. (The standard does specify that all side effects on arguments take effect before a function is called, but that would not help in the call to printf above.) The moral is that writing code that depends on order of evaluation is a bad programming practice in any language. Naturally, it is necessary to know what things to avoid, but if you don’t know how they are done on various machines, you won’t be tempted to take advantage of a particular implementation. Chapter 3 - Control Flow The control-flow of a language specify the order in which computations are performed. We have already met the most common control-flow constructions in earlier examples; here we will complete the set, and be more precise about the ones discussed before. 3. 1 Statements and Blocks An expression such as x = 0 or i++ or printf(...) becomes a statement when it is followed by a semicolon, as in x = 0; i++; printf(...); In C, the semicolon is a statement terminator, rather than a separator as it is in languages like Pascal. Braces { and } are used to group declarations and statements together into a compound statement, or block, so that they are syntactically equivalent to a single statement. The braces that surround the statements of a function are one obvious example; braces around multiple statements after an if, else, while, or for are another. (Variables can be declared inside any block; we will talk about this in Chapter 4.) There is no semicolon after the right brace that ends a block. 3. 2 If-Else The if-else statement is used to express decisions. Formally the syntax is if (expression) statement1 else statement2 where the else part is optional. The expression is evaluated; if it

is true (that is, if expression has a non-zero value), statement1 is executed. If it is false (expression is zero) and if there is an else part, statement2 is executed instead. Since an if tests the numeric value of an expression, certain coding shortcuts are possible. The most obvious is writing if (expression) instead of if (expression != 0) Sometimes this is natural and clear; at other times it can be cryptic. Because the else part of an if-else is optional, there is an ambiguity when an else if omitted from a nested if sequence. This is resolved by associating the else with the closest previous else-less if. For example, in if (n > 0) if (a > b) z = a; else z = b; the else goes to the inner if, as we have shown by indentation. If that isn’t what you want, braces must be used to force the proper association: if (n > 0) { if (a > b) z = a; } else z = b; The ambiguity is especially pernicious in situations like this: if (n > 0) for (i = 0; i < n; i++) if (s[i] > 0) { printf("..."); return i; } else /* WRONG */ printf(" error -- n is negative"); The indentation shows unequivocally what you want, but the compiler doesn’t get the message, and associates the else with the inner if. This kind of bug can be hard to find; it’s a good idea to use braces when there are nested ifs. By the way, notice that there is a semicolon after z = a in if (a > b) z = a; else z = b; This is because grammatically, a statement follows the if, and an expression statement like ‘‘ z = a;’’ is always terminated by a semicolon. 3. 3 Else-If The construction if (expression) statement else if (expression) statement else if (expression) statement else if (expression) st