Programming in C

Rex Jaeschke

Programming in C

Edition: 4.0

The training materials associated with this book are available for license. Interested parties should contact the author at the address below.

Please address comments, corrections, and questions to the author:

Rex Jaeschke
2051 Swans Neck Way
Reston, VA 20191-4023
+1 (703) 860-0091
+1 (703) 860-3008 (fax)
www.RexJaeschke.com
rex@RexJaeschke.com

Much of the information in this book has been published previously, as follows:

1. A series of monthly columns called "Let's C Now" in *DEC Professional*, Professional Press, running from 1984–1995.
2. A two-volume set of reprints called "Let's C Now", Professional Press, 1986 and 1987, ISBN 0-9614729-2-8 and 0-9614729-3-6.
3. The book "Mastering Standard C", Rex Jaeschke, Professional Press 1989, ISBN 0-9614729-8-7.
4. The book "Mastering Standard C", Rex Jaeschke, 2e, CBM Books 1996, ISBN 1-878956-55-8.

The glossary of terms from "Mastering Standard C", 1e, was spun-off and expanded into its own book, "The Dictionary of Standard C", Prentice Hall 2001, ISBN 0-13-090620-4. This dictionary can be viewed on-line, free of charge, at www.prenhall.com/jaeschke. An earlier version of this dictionary, now out of print, was published by Professional Press Books 1991, ISBN 0-07-707657-5.

Preface

Welcome to the world of C. Throughout this book, we look at the statements and constructs of the C programming language. Each statement and construct is introduced by example with corresponding explanations, and, except where errors are intentional, the examples are complete programs or subroutines that are error-free. I encourage you to run and modify these examples, because the only realistic way to master a language is to write programs in it, run them, and debug them.

This book was written with teaching in mind. It is intended for use both in a classroom environment as well as for self-paced learning.

Dennis Ritchie developed C around 1972 at AT&T's Bell Laboratories. C evolved from the languages CPL, BCPL, and B, in that order. The UNIX operating system, another Bell Labs development (designed by Ken Thompson and Ritchie), then was rewritten in C. (Previously, UNIX was written in assembler and B.) UNIX and C have been closely associated ever since, and every UNIX and UNIX -like operating system includes a C compiler. Today, C has a life quite separate from UNIX, with implementations of C available for every mainstream hardware and operating system platform.

C is a general-purpose, high-level language. From a grammatical viewpoint, C is a relatively simple language having some 37 keywords. Because of the power of the basic statements and constructs, and the fact that the programmer can effectively extend the language by using callable functions while still maintaining portability, there is little need for language extensions especially considering the fact that C supports calls to procedures written in other languages. However, extensions do exist in some compilers and vary from one vendor to another. You should be aware of any nonstandard conventions in production compilers you use or evaluate. Most people learn a language by using one particular dialect of it. As a result, they are often unaware when they are using an extension.

Depending on their language backgrounds, programmers new to C may initially find programs hard to read because, traditionally, most C code is written in lowercase. Lower- and uppercase letters are treated by the compiler as distinct. In fact, C language keywords must be written in lowercase. Keywords are also reserved words.

C supports a structured, modular approach to programming using callable subroutines, called *functions*. Source files can be compiled separately, with external references being resolved at linktime.

C lends itself to writing terse code. However, there is a fine (and subjective) line between writing code that is terse and code that is cryptic. It is easy to write code that is unreadable and, therefore, unmaintainable. However, with care and a small dose of discipline, you can produce nicely formatted code that is easy to read and maintain. However, good code doesn't happen automatically—you have to work at it. Throughout the book, I make numerous comments and suggestions regarding style. Perhaps the best advice I can give in that regard is "Remember that the poor fool who has to read your code in the future just might be you!" Above all, be overt and be consistent.

Programming in C

Numerous "Tips" and "Style Tips" have been added throughout to highlight important suggestions. Considerable attention has been paid to using and teaching a consistent, popular, and overt programming style. Both kinds of tips are highlighted so they stand out. For example:

> **Tip:** Avoid initializing enumeration constants explicitly or making duplicates unless you have good reason to do so. Assigning a range of values that is non-continuous or that contains duplicates significantly reduces the set of operations that can meaningfully be performed.

> **Style Tip:** se liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it. C is not all things to all people; nor does it claim to be. For many applications, assembly language, Pascal, COBOL, FORTRAN, and BASIC, for example, will do just fine. In any event, compared to other high-level languages (including Java and C#), C is a relatively expensive language to learn and master.

Reader Assumptions

This is *not* a first course in programming. (Nor do I recommend C as a first programming language.)

I assume that you know how to use your particular text editor, C compiler, and debugger. Comments on the use of these utility programs will be limited to points of particular interest to the C programmer.

To fully understand and exploit the material, you should be conversant with the following topics:

- The basic purpose of a compiler and linker.
- Number system theory.
- Bit operations such as AND, inclusive OR, exclusive-OR, complement, and left- and right-shift.
- Data representation.
- Communication between procedures by passing arguments and/or by returning a value, as well as via global variables.
- Use of single- and multi-dimensional arrays.
- Creation and use of sequential files and how to do formatted and unformatted I/O.
- Basic data structures such as linked lists.

If your programming background is in some procedural language such as PL/I, Pascal, FORTRAN, or BASIC, you'll need to read the whole book closely and do all the exercises. As for those of you having only a COBOL background, you'll have some catching-up to do with respect to thinking in binary versus decimal when it comes to data representation, as well as the use of functions and their associated argument passing and value returning machinery.

Limitations

This book covers almost all of the C language. It also introduces the core class library. However, a very small percentage of library facilities are mentioned or covered in any detail. The Standard C library contains so many functions that whole books have been written about that subject alone.

This book is directed at teaching the C language proper, by writing simple stand-alone applications, since without a thorough knowledge of the language, you will not be able to understand and implement other, more advanced, solutions.

GUI, calling non-C routines, threading, inter-process communications, operating system-specific, and a number of other advanced features are outside the scope of this text.

Presentation Style

The approach used in this book is different from that used in many other books and training courses. Having developed and delivered programming language training for some 14 years, I have found that the best approach for my students is an incremental one. I avoid introducing things that are unnecessary at any given time, thus making examples small, simple, and focused. Many books use GUI and numerous non-trivial library facilities in the first few examples, and certainly in the first chapter. I do not care for this approach, either as a reader or as an educator. Instead, I prefer the student to have an excellent chance of understanding and absorbing small amounts of new material, and reinforcing it with lab sessions, as they progress. The intent here is to eliminate any chance of their being overwhelmed, provided, of course, they meet the prerequisites.

Different styles work for different teachers and different students. I do not suggest that my approach is better than any other is; I simply know that my approach works well, and has formed the basis of my successful seminar business.

Exercises and Solutions

The programs shown in the text are available electronically in a directory tree named Source, where each chapter has its own subdirectory, within which each program has its own subdirectory. For example, the source code for the program called ba04 in the "Basics" chapter can be found in the following directory hierarchy: Source, Basics, ba04. By convention, the names of C source files end in .c.

Each chapter contains exercises, some of which have the character * following their number. For each exercise so marked, a solution is provided electronically in a directory tree named Labs, where each chapter has its own subdirectory, within which each program has its own subdirectory.[1] For example, lab solution lbba01 in the "Basics" chapter has the following fully qualified directory hierarchy: Labs, Basics, lbba01.

Exercises that are not so marked have no general solution and may require experimentation or research in an implementation's documentation.

---

[1] The solutions are only available to licensees of these materials when they are used in formal training scenarios.

Programming in C

You are strongly encouraged to solve all exercises in one section before continuing to the next. Also, invent your own exercises as you go and be inquisitive; don't be afraid to experiment. Try to understand why the compiler gives you each error or why a program fails at run time.

Program Behavior

According to the C Standard, for correct, well-formed programs, almost all behaviors are well defined and predictable. However, in certain cases, an implementation can chose the behavior it deems best, and this behavior need not be the same as that exhibited by other implementations. Such behavior is called *implementation-defined*, and must be documented by each implementation. For example:

> **Implementation-Defined Behavior:** The range of values that can be stored in an object of a given arithmetic type.

In other cases, an implementation can chose whatever behavior it wants at that time *without* needing to reproduce or document that choice. Such behavior is called *unspecified*. For example:

> **Unspecified Behavior:** Whether two string literals containing the same characters are stored in distinct arrays.

A third category is *undefined behavior*, which can result from situations for which the standard imposes no requirements or is otherwise silent. For example:

> **Undefined Behavior:** Subscripting an array with an out-of-bounds index.

Clearly, we must be aware of implementation-defined and unspecified behaviors when writing code that is to be ported across different platforms. And we should always avoid relying on undefined behavior.

Far too many people believe that just because their C code has worked for a long time, it must be correct. When next your compiler is upgraded, the behavior it exhibits with "undefined behavior" constructs may well be different (as it is permitted to be), resulting in old code breaking. In some cases, simply using a different combination of compiler options with the same compiler can change the way the compiler works internally. You should never try to figure out how your compiler works in "undefined" cases. Any test case you write will be so trivial as to be meaningless. And having 10, 100, or even 1,000 test cases exhibit the same "undefined" behavior is still no guarantee that the next test won't behave differently.

The Status of Standard C

The history of the standardization of C is as follows:

- C89 – The first ANSI C standard, ANSI X3.159-1989, was produced in 1989 by the U.S. committee X3J11.
- C90 – The first ISO C standard, ISO/IEC 9899:1990, was produced in 1990 by committee ISO/IEC JTC 1/SC 22/WG 14 in conjunction with committee X3J11. C90 was technically equivalent to C89.

- C95 – An amendment to C90 was produced in 1995 by committee WG 14 in conjunction with the U.S. committee X3J11. The additions included digraphs, the header `iso646.h`, and many multibyte and wide-character functions via the headers `wchar.h` and `wctype.h`.
- C99 – The second edition of the ISO C standard, ISO/IEC 9899:1999, was produced in 1999 by committee WG14 in conjunction with the U.S. committee INCITS/J11 (formerly X3J11). The additionsd included a few language features, a number of headers, and many library functions. Throughout its development, C99 was commonly referred to as C9x.

The C standards committee is currently working on various maintenance issues and informative Technical Reports.

Electronic copies of C99 can be purchased for US$18. (For more information, see www.ansi.org or www.iso.ch.)

Acknowledgments

Many thanks to those people who reviewed all or part of this book. In particular, students in my C seminars provided useful feedback and located numerous typographical errors.

Rex Jaeschke, September 2009

1.   The Basics

In this chapter, we will learn about a number of fundamental constructs and language elements.

1.1   Basic Program Structure

A correct source program is made up of an acceptable combination of tokens, where a *token* is the smallest possible unit of source. C has five different kinds of tokens: keywords, identifiers, literals, operators, and punctuators.

For the most part, C is a free-format language, with space between tokens being optional. However, in some cases, some kind of separator is needed between tokens so they can be recognized the way they were intended. White space performs this function. *White space* consists of one or more consecutive characters from the following set: space, horizontal tab, vertical tab, form-feed, and *new-line* (entered by pressing the RETURN or ENTER key). In a source file, an arbitrary amount of white space is permitted before the first token, between any two adjacent tokens, and after the last token.

Let's look at the basic structure of a C program, which writes a welcome message to the console (see directory ba01):

```
/*‒
This program is a simple example of C.
‒*/

/*1*/ #include <stdio.h>

int main()              /* start of the program */
{                       /* beginning of function body */

/*2*/   printf("Welcome to C\n");

/*3*/   return 0;       /* terminate with success */
}                       /* end of a function body */
```

A delimited comment is one surrounded by /* and */, and can be used on part of a source line, as a whole source line, or it can span any number of source lines. C99 added C++'s line-oriented comments, which begin with // and continue until the end of the same source line.[1] A comment of either kind is treated as a single space. A delimited comment can occur anywhere white space can be used. A line-oriented comment can only appear at

---

[1] At the time of writing, support for C99 was not yet widespread. As such, line-oriented comments are not used in this book.

the end of a source line. Although comments of the same kind do not nest, each kind of comment can be used to disable a source line containing the other.[1]

A C program consists of one or more functions that can be defined in any order in one or more source files (or *translation units*, as Standard C calls them). A program must contain at least one function, called `main`, and this function's name must be spelled using lowercase letters.[2] This specially named function indicates where the program is to begin execution.[3] The `int` keyword preceeding `main` indicates that `main` will return a result of type `int`, as discussed below.

The parentheses following the function name `main` surround the function's parameter list.[4] The parentheses are required, even if no arguments are expected.

The body of a function is enclosed within a matching pair of braces. All executable code must reside within the body of some function or other. Statements are executed in sequential order unless branching or looping statements dictate otherwise. A program terminates when it returns from `main`, either by dropping into the closing brace of that function—which acts as an implicit `return` statement—or via an explicit `return` statement.

We'll learn about the `return` statement in §1. However, why return a value from `main`? Many programs either are invoked at the operating system's command-line level or are spawned by another program. In either case, when a program terminates, it has the ability to return one piece of information to the program that invoked it. Perhaps it returns a status code or a count of the number of transactions processed. We refer to this returned value as the program's *exit status code*. The value used and its meaning are the business of the programmer. Unless otherwise stated, all `main` functions in this book contain the statement `return 0;`. The value zero has no special significance, although on some systems it is interpreted as some form of success code.

Case 1[5] provides access to the I/O library, and case 2 outputs the welcome text to the console. These cases are discussed in detail in §1.3.

> **Style Tip:** Use liberal amounts of white space to improve program readability. The compiler discards all white space, so its presence has no effect on program execution. Apart from separating tokens, white space exists solely for the benefit of the reader. If you can't read the code, you surely won't be able to understand it.

We can write any given correct program in an infinite number of ways simply by using different amounts and kinds of white space (including comments). Obviously, there are very good, overt styles and very bad, cryptic styles. Then there is the large and subjective gray area in between.

---

[1] Refer to §6.4 to see a novel way to "comment out" a source line containing a comment.
[2] While mixed- or uppercase spellings of `main` might be recognized by some systems, this is undefined behavior.
[3] Some environments require a different entry-point name. For example, certain Microsoft Windows programs begin execution at `winmain` instead.
[4] A function uses parameters to declare what it is expecting to be passed, via an argument list, at run time.
[5] Throughout this book, many code examples contain source lines with leading delimited comments of the form /*n*/, when *n* is a number. Such lines are referred to as "cases", with /*1*/ being case 1, /*2*/ being case 2, and so on.

> **Style Tip:** Be overt; pick a style that isn't too far outside the mainstream, and stick with it. Above all, be consistent. And remember, the style you develop when learning a language is the one with which you will likely stay, so give some thought to programming style from the outset.

The style used in this book is very common. If you choose to adopt a different style, note that no matter how great you think your particular style is, every minute you spend arguing about whose style is superior and why—or worse yet, changing other people's code to your style—is generally time wasted. If no corporate-wide or group style guide exists, and the people on a multi-person project cannot agree on a common style, the project manager should dictate one.

> **Exercise 1-1*:** Compile and link the empty program listed above, and look at the size of the resulting executable file on disk; it may be surprisingly large. Look at the listing produced by the linker and see what goes into a seemingly "empty" program. (See lab directory lbba03.)
>
> **Exercise 1-2*:** Try compiling, linking, and executing a main program whose name is something like `Main` or `MAIN`. If either of these works, use `xyz` instead and try again. (See lab directory lbba04.)
>
> **Exercise 1-3*:** Using /* and */, comment out a block of code that already contains this kind of comment, and see how your compiler reacts. (Some compilers actually do support nested comments as an extension.) (See lab directory lbba05.)
>
> **Exercise 1-4*:** What happens if you leave off the closing */ from a comment? (See lab directory lbba06.)

Let's move on to the more common case of a program having multiple functions (see directory ba02):

```c
/* C program with two functions */

void compute()
{
        /* ... */
/*1*/   return;         /* redundant statement */
}

int main()
{
/*2*/   compute();      /* call function 'compute' */

        return 0;
}
```

C supports modularization via *functions*. A source file can contain one or more functions, defined in any order. (That said, in this example, it is no accident that function `compute` is defined before function `main`. In §1, we'll learn how to define functions in any order as well as how to put them in separate source files.) Unlike some languages, in C, function definitions cannot be nested. That is, each function's definition must be outside the

3. Arrays and Strings

In this chapter, we will learn how to define and use arrays. While only a few types are used in the examples, the principles can be applied to arrays of objects of any data type, including user-defined types. We'll also look at string manipulation, and some character testing functions from the standard library.

3.1   Introduction

There are four main differences between using arrays in C and in most other languages.

5. In array definitions and subscripts, square brackets are used instead of parentheses. This is a minor issue, and a compiler will produce an error if we inadvertently use parentheses instead.
6. Each dimension in a multidimensional array is written with its own set of brackets. This allows an array to be subscripted with less than the maximum number of dimensions, something not possible in most other languages.
7. Array elements begin at subscript 0, not 1. This may or may not be a major problem, depending on one's language background. It can result in "off by one" errors. For example, if we access an array of 10 elements by subscripting from 1 to 10, we miss element 0 and we access the nonexistent element 10.

> **Undefined Behavior:** Attempting to access an out-of-bounds array element.

And to make it really interesting, for the most part, C compilers cannot easily enforce array bounds checking, so they don't.
8. Subscripting involves the use of [ ]. In this context, [ ] represents an operator, so it is subject to the rules of precedence and associativity.

> **Tip:** Programmers new to C often forget that for an array with a dimension of size n, the range of valid subscripts for that dimension is 0 through n-1.
>
> **Tip:** The following expressions exhibit undefined behavior because the order of evaluation across the assignment operator is undefined: `a[i] = b[i++]` and `a[i++] = b[i]`. Similarly, `x[i][i++]` and `x[++i][i]` exhibit undefined behavior because the order of evaluation across the subscript operator is undefined.

Some array definitions and subscript expressions follow (see directory ar01):

```
int main()
{
        float f[10];              /* [0] … [9]        */
        long l[3][5];             /* [0][0] … [2][4] */
        int i, j, k;

        for (k = 0; k < 10; ++k)
        {
                f[k] = 0.0F;
        }

        for (i = 0; i <= 2; ++i)
        {
                for (j = 0; j <= 4; ++j)
                {
                        l[i][j] = 10;
                }
        }

        return 0;
}
```

The number of elements in each dimension of an array declaration must be a compile-time integer constant expression with a value greater than zero. In particular, note that, unlike most other languages, the dimension expression can contain any arithmetic operators that make sense with integer constants. For example, f could have been declared using `float f[5 + 5]`.[1]

Although variable-size arrays are not permitted by the language,[2] they can be implemented at run time by using various library functions.[3]

In C, arrays are stored in row-major order. That is, elements are stored in memory such that the right-most subscript varies the quickest, like BASIC, COBOL, and {Pascal}, but unlike FORTRAN.

C places no limit on the number of dimensions; we are limited only by the amount of memory available.

If an array name is defined with the `const` qualifier, all of its elements are `const`-qualified.

Note that in the example above, the braces in the single and nested `for`s are all optional, since the body of each is only a single statement.

> **Exercise 3-1*:** Define an `int` array with only one element. Initialize it to the value 100, and display its value using `printf`. (See labs directory lbar03.)

---

[1] The importance of this capability will be seen in §6.2.
[2] Strictly speaking, this isn't true. C99 added variable-length local arrays. However, these are not discussed in this book.
[3] This capability is discussed in detail in §7.11.

Exercise 3-2*: Define an `int` array of 10 elements. Display the value of elements 15, 100, and -5, all of which are nonexistent. Do you get any compilation or run-time errors? If so, can you explain them? (See labs directory lbar04.)

3.2   Initialization

An array is an *aggregate*; that is, an object containing subobjects, each of which has its own value. While we can set the value of each element individually via assignment, that approach is tedious. Instead, we can use an *initializer*, which consists of a brace-delimited list of expressions; for example (see directory ar05):

```
int counts[5] = {10, 35, 56, 76, 12};
double value[3] = {1.2, 2.3, 3.4};

enum Color {red, brown, green, blue};
enum Color hue[5] = {green, blue, red, brown, red};
```

When we are initializing a multidimensional array, each dimension has its own initializer list. For example:

```
int iarray1[2][3] = {
        {1, 2, 3},
        {5, 4, 3}
};
```

The array definition is read as "`iarray1` is an array containing two elements, each of which is an array containing three elements". Therefore, the large initializer list contains two subinitializer lists, each of which corresponds to a row. A 3-D array is initialized in a similar manner:

```
int iarray2[2][3][2] = {
        {{1, 2}, {4, 5}, {2, 3}},
        {{6, 1}, {9, 3}, {0, 2}}
};
```

The array definition is read as "`iarray2` is an array containing two elements, each of which is an array containing three elements, each of which contains two elements".

Some languages, notably FORTRAN, provide a repetition count when identical consecutive initializing expressions are required. C does not provide such a capability. If we wish to initialize each element of a 100-element array with the value 1, for example, we must specify an initializer list with 100 expressions containing that value.

C99 enhanced initialization with the addition of *designated initializers* (which are not discussed further in this book).

An array initializer list need not contain values for every element. For example:

7.   Pointers and Addresses

So far, we have learned how to do things in C that we already know how to do in some other language. In this chapter, we will be exposed to those capabilities of C that make it different from most high-level languages.

7.1   Introduction

Without a doubt, a difficult part of C to master is pointers. For most high-level language programmers, the idea of using addressing directly and defining variables—called *pointers*—that contain addresses is quite foreign. And even after these programmers come to grips with the idea, they often have considerable difficulty in dealing with the syntax.

To many programmers, pointers seem unnecessary. After all, they have implemented thousands of lines of production code over many years and they have never had the need to even know about, let alone use, pointers. Why then must they master them to use C? That is a reasonable question and one that we will address.

7.2   A Conceptual Model of Memory

Before we can talk much about addressing and pointers, we need a conceptual model of how memory is organized when a program is loaded for execution. Consider a program that contains the following fragment:

```
void f()
{
        static int i = 10;

        statement(s)
}
```

When this program is loaded into memory for execution, Figure 7–1 below indicates how things might be organized, with regard to the variable i and the function f. (The exact way in which a system represents this may vary. However, the concepts suggested by this model still apply.)

Figure 7–1: Code and Data Mapping

1000	int i 10
4000	f()
4001	{
4098	
4099	}
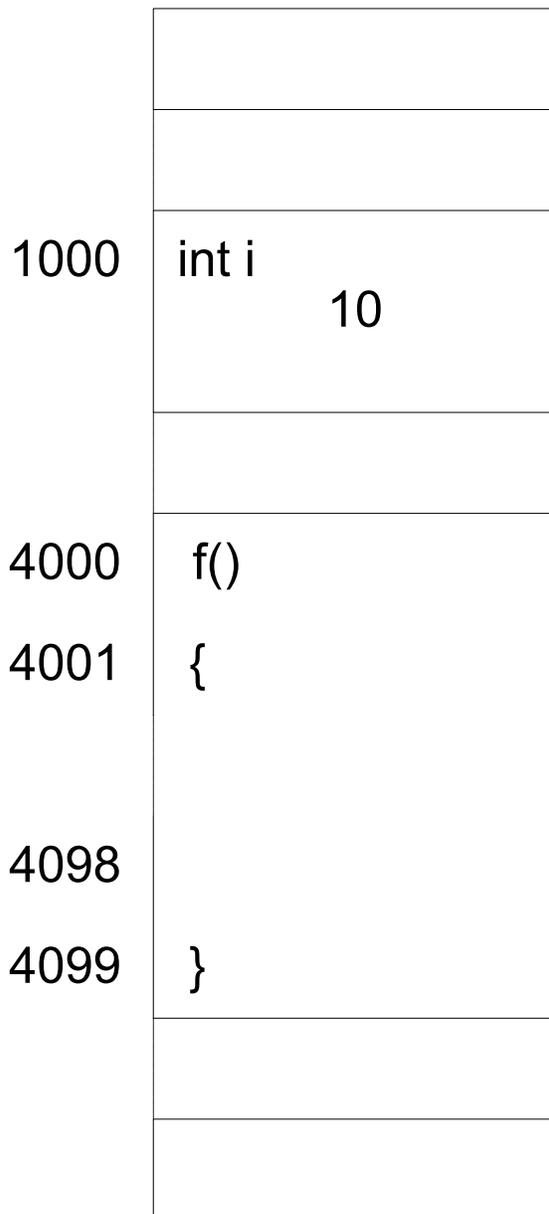
Memory is organized into an array of cells, with each cell having a unique location, which we shall call an *address*. On machines whose memory is *byte-addressable*, each byte has its own address. On machines that are *word-addressable*, each word has its own address. In the latter case, bytes do not physically exist; characters must be packed into words, and the address of a packed character is the address of its parent word and the character offset within that word.

Many machines use addresses that really are unsigned integers. Others use addresses that are signed integers. Still others use a segmented address that is made up of two parts: a base address and an offset. For the purposes of our discussion, we will make no assumptions about how physical memory really is addressed on any particular machine; that is unimportant in our model.

Function `f` contains a `static int` object, `i`. Being static, this variable typically is allocated memory by the compiler and/or linker. In any event, memory is allocated for it before `main` begins execution. `i` occupies one or more consecutive locations in memory, starting at address 1000. We say then that the address of `i` is 1000. (The addresses used in this discussion have been picked arbitrarily. For actual memory placement details, use your debugger or consult the listing file produced by your linker.)

Once the compiler and linker have done their job, all variable and function names have disappeared, and all references to them have been reduced to references to their corresponding addresses. And yes, even functions have addresses. In Figure 7–1, the machine instructions for the executable code in function `f` were allocated memory locations 4000–4099.

The reason for all this is that a CPU is a very simple-minded machine. It doesn't know anything about abstractions such as variables and functions. What it does know is how to go to a specific address in memory to get or put a single- or multibyte object. It also knows how to branch or jump to an address and begin executing instructions at that location. Therefore, the principal job of a compiler and linker, in combination, is to reduce source code to an ordered set of machine instructions, many of which deal with addresses.

Ultimately, the CPU deals with addresses. However, that doesn't necessarily mean we as programmers need to deal with them. In fact, most languages provide no way to deal with addresses directly. However, C does, and while we can try to avoid using them directly, addresses are produced and used all over the place in things as fundamental as uses of `scanf` and `printf`. There is no escaping addresses in C; for better or worse, addresses simply are an integral part of C programming.

As discussed above, objects and functions are referenced via their addresses. And if we change our code in certain ways, when next we compile and link, these objects and functions may be located at different addresses. Therefore, we are rarely, if ever, interested in the particular address at which something is located. In fact, the addresses used during any particular execution of a program are relative to that execution only. Consider the case of an automatic variable. It is allocated memory at run time, typically on some stack. If we call its parent function from several different places, it is almost certain that for each of its lifetimes, the variable will live at different locations. (This would certainly be true for automatic variables defined in functions that are called recursively.)

If addresses can easily change during execution or over the life of an application, why are we interested in them? If we know where something lives, we can remember that information and, later on, access it indirectly through that saved address. Accessing objects and functions in this manner provides a very powerful abstraction tool, and one that we will explore in the next section.

In Figure 7–2, our memory model has been extended to where we can save an object's address. The address of `i` is 1000. If we can find out `i`'s address, we can use it to access `i` indirectly. We might also want to save that address in memory for future reference. We do this by creating a variable capable of storing the address of `i`. Let us call it `pi`, for "pointer to `int`". Let us assume that the pointer variable is located at address 1100.