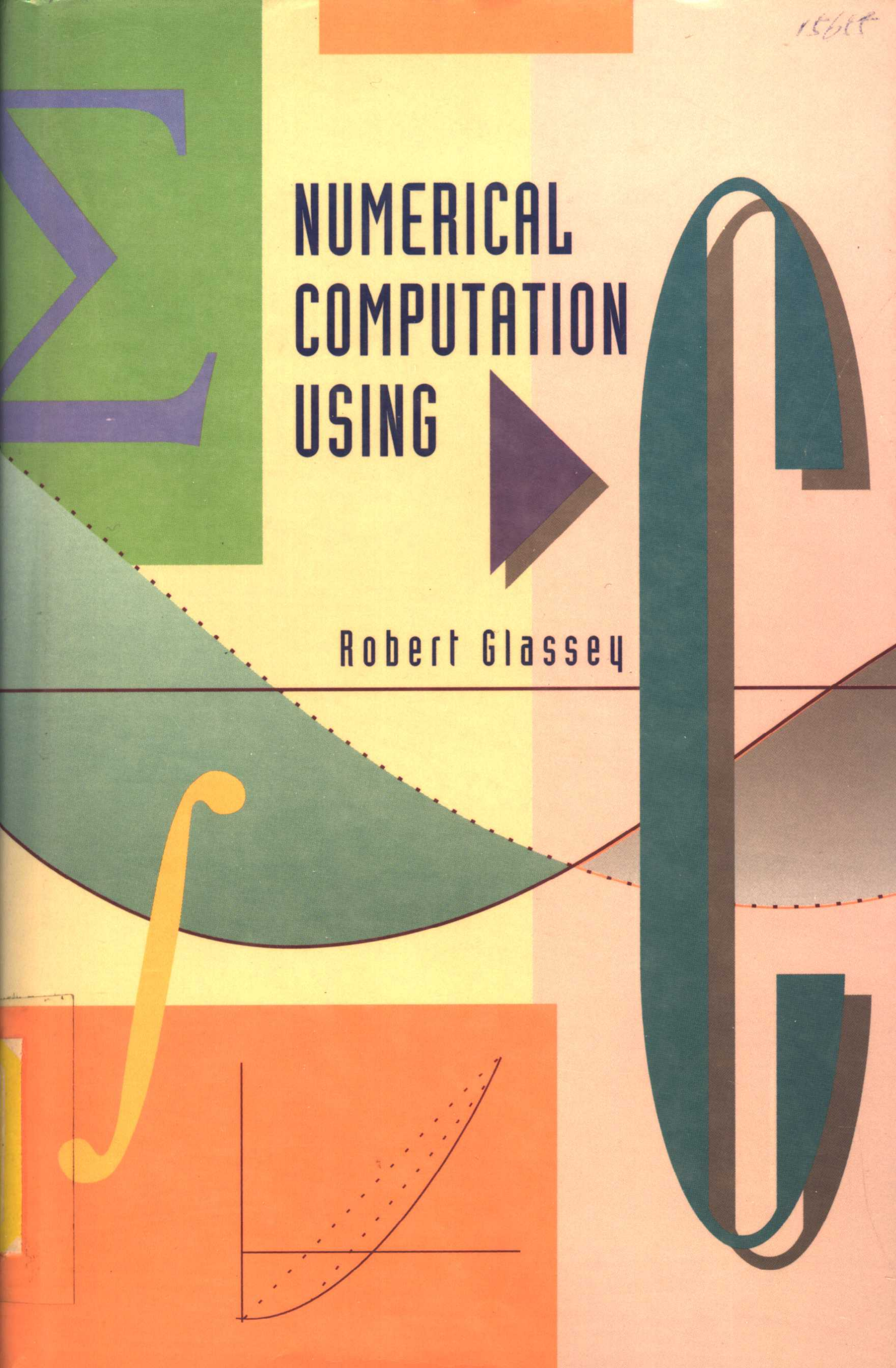
15688

NUMERICAL COMPUTATION USING C

Robert Glassey



NUMERICAL COMPUTATION USING C

ROBERT GLASSEY

Department of Mathematics
Indiana University
Bloomington, Indiana

ACADEMIC PRESS, INC.
Harcourt Brace Jovanovich, Publishers
Boston San Diego New York
London Sydney Tokyo Toronto

This book is printed on acid-free paper. ♾

ACADEMIC PRESS, INC.
1250 Sixth Avenue, San Diego, CA 92101

United Kingdom edition published by
ACADEMIC PRESS LIMITED
24-28 Oval Road, London NW1 7DX

Library of Congress Catalog Card Number: 92-33037

ISBN 0-12-286155-8

Printed in the United States of America
92 93 94 96 BC 9 8 7 6 5 4 3 2 1

ACKNOWLEDGEMENTS

I wish to thank my wife Betsy for enduring many hours alone while I constructed this book. My son Tom deserves sincere thanks for not crashing my hard disk while (patiently?) waiting for his bedtime story ....

I wish to specially thank Bishara Shamee from whom I learned much about *C* and UNIX.

To my many friends and colleagues who offered advice and suggestions, I am most grateful. They include Kelly Alvey, Michele LeBlanc, John Willems, Scott Zasadil and Bill Ziemer.

Thanks also to Ms. Jenifer Swetland and the staff of Academic Press with whom it was a pleasure to work.

TABLE OF CONTENTS

CHAPTER 0

INTRODUCTION

There are several reasons for my writing this book. Firstly, in numerical analysis classes at the senior–first year graduate level, I find that students who claim to know *C* do not in fact know it very well. In particular, pointers, arrays, dynamic memory allocation, etc., are troublesome.

Secondly, from my own experience of learning *C*, I found that the examples in existing texts were mostly system–oriented; few if any involved scientific computation. In 1990 I was shocked to read in *PC Magazine* a statement by a veteran programmer to the effect that, despite many years in the profession, he had never written a single computational program. Although there are many nice references on *C*, I do not know of any at the introductory level which are written from the viewpoint of a mathematician.

Furthermore, as I learned *C*, I wished for a mathematically oriented reference in which I could look things up quickly. It is certainly true that the recent excellent publication *Numerical Recipes in C* almost fits the bill; there it is assumed that the reader knows *C* already. In this book I will assume that the reader already knows some language and is familiar with the uses of loops, if–then–else statements, etc.

This book is not a text on the *C* language proper, nor is it a text on numerical analysis. It is intended to be a guide for learning *C* from the viewpoint of numerical analysis. As such it is a hybrid, perhaps to be used as a supplement in a course in numerical analysis. I intend it to be more or less brief and inexpensive, so that students can readily afford it. Indeed, I quote from [KR]: "*C* is not a big language, and is not well–served by a big book."

How does one learn a new language? The answer is: by reading a book, by studying the code of others, and by sitting down at your terminal and enduring the edit–compile–run cycle until you get it right. The intended audience is probably split into two groups: one which uses a PC under DOS (or a Macintosh) to develop programs and another which uses UNIX. It is well-known that large–scale computation under DOS is impossible. Nevertheless program development on a PC is very convenient, and there are several nice *C*–compilers available. On the PC–level, I use the Microsoft "Quick *C*" (v. 2.5) compiler. For UNIX, I have in mind *gcc*, the GNU *C*–compiler. Both understand ANSI and are a pleasure to use. I omit the Borland product and all others simply because I have never used them.

While graphics are built into "Quick *C*," one can use *gcc* to write the results of a computation to a file and then feed the file to, say, GNU-PLOT, another GNU product which swallows files and produces nice two–dimensional graphics. The most recent version of this program (v. 3.0) can display three–dimensional graphs as well. These GNU products are excellent, the price is right and they work as advertised. Since I have not yet contributed to the Free Software Foundation, I feel this "plug" is warranted! Nothing in this book is compiler–specific; all programs should run on nearly any *C*–compiler, modulo a few minor changes.

In recent years several magnificent programs have appeared in the mathematical area: Mathematica, Macsyma, Maple, Derive, Gauss, Matlab, etc. In view of all of this power, should you still consider learning a language? The answer is an unqualified YES. The nature of this business is so specialized that you will not always be able to get one of these programs to do what you want. Moreover, the computation of solutions to large–scale Partial Differential Equations is an ad–hoc process for which the ability to write your own code is indispensable. If you require only small computations, BASIC is easy to learn and use. In my opinion, Microsoft "Quick Basic" v. 4.5 is a tremendous program and is to be highly recommended. For larger computations, of course, Fortran has been the standard in scientific computation, and superb libraries are available. These libraries are now available in *C*.

C is a general–purpose language which has been traditionally used in systems programming. Indeed, UNIX is written in *C*. *C* can be adapted to a broad spectrum of applications and boasts wide choices of data types, e.g., pointers, structures, etc. It contains a large set of operators and control devices, yet it is a "small" language. The standard run–time libraries contain code for dynamic memory allocation, input/output, etc. Moreover, *C* is to be recommended for its portability, efficiency and elegance, and is certainly in favor in academics and in the real world as well. There are some drawbacks from the scientific–computation point of view. For example, there is no built–in exponentiation function, nor is there a built–in facility for complex arithmetic. Of course, these computations are still possible in *C*, but a speed penalty is incurred.

A word or two about the plan of the book: Chapter 1 is an overview of the C language, put in terms of *mathematical* examples. Chapter 2 deals with the uses of *pointers* , their applications to memory allocation for vectors and matrices, etc. In Chapter 3 we cover certain smaller topics which did not seem to fit elsewhere, and some of the fine points. The last chapter of the book covers special topics, such as linear algebra, differential equations, etc. My intent is for you to find some nontrivial example programs here, as well as some of the mathematical background.

An indication of the proofs of those results which are not too detailed will be given. In my opinion, the only way to really understand an algorithm is to first prove it converges, and then to code it. Of course for most real problems encountered in practice, the idea of giving a rigorous proof may not be achievable, but there is nothing wrong with professing this as a goal.

As for background, that of an advanced undergraduate in mathematics or the physical sciences should be sufficient. An extensive knowledge of real analysis is not required, but familiarity with standard topics (such as the convergence of a sequence, the Mean–Value Theorem, Taylor's Theorem, etc.) is assumed.

Here are some comments about the programs. Undoubtedly there will be some errors, typos, etc. Rather than strive for the slickest possible coding, I have tried to make the programs simple and readable. Thus I do not claim that these programs are the best available, nor that they are optimized. If you can understand the coding of the basic form of an algorithm, then later on in life when you use a "canned program" you may feel fairly confident that you understand what is going on. (There is perhaps an analogy here to the study of special functions.) Each C function used in a program is (at least at the beginning of the book) explicitly included at the top of the file. While this is repetitious, it renders most of the programs self–contained. I have kept the number of special files to be "included" to an absolute minimum for simplicity. I encourage you to experiment with the programs, and to alter them to suit your needs. I would be pleased to hear about errors, bugs, etc.

The "tolerance" 5×10^{-20} is (arbitrarily) used to test a denominator before a division is performed. The "stopping criterion" in iteration loops varies in the programs. This tolerance may have to be adjusted (i.e., relaxed) if the data of a particular problem are "large." When the elements of a symmetric matrix are to be read from files, we always construct the data files by entering the first row, then the second row (from the diagonal to the right), etc. For the sake of uniformity we use *double precision* in most of the book (with the exception of the first few programs). This is consistent with calls to the functions in `math.h`, but can be easily changed.

A major topic for which C is employed is *string handling*. This is not discussed in this book, since the emphasis is on scientific computation. The manipulation of strings can be tricky; please consult the canonical reference [KR] for details. A related reference is [HS] which contains topic-oriented material on the *C* language proper. Both of these books contain descriptions of the standard libraries and are to be highly recommended.

There are other omissions in this book, e.g., unions, linked lists, the use of the bitwise operators, etc. Moreover, not all of the properties of the built-in functions (e.g. `printf, scanf`) are fully exposed. Therefore you will not become a *C* master by reading this book alone. Some topics have been omitted for considerations of length, others because I have never used them. Thus I urge you to consult other references for a more complete picture of *C* as a language; here we will use and study *C* with a specific goal in mind.

The numerical integration of Partial Differential Equations is an extremely interesting subject and is at the forefront of modern research in applied mathematics. The sheer size of many problems is daunting. Furthermore, the development of algorithms to accurately handle *nonlinear* phenomena is in its infancy. For these reasons a practitioner or student in the area of applied mathematics simply must be able to code his or her own work. The *C* language is a perfect environment for this and, modulo mutations, it is sure to be around for many years. A final quotation from [KR], which I have found to be appropriate, is "*C* wears well as one's experience with it grows."

SOME COMPILATION/RUN-TIME TIPS

1. Using Microsoft QC 2.5 on a PC

After entering your source code with the editor, just press F5 to compile and run. In my `autoexec.bat` file, I include the line

```
set cl=qcl /AS /Ol
```

This uses the small memory model and optimizes loops. From the command line in the QC 2.5 directory, use `qcl filename.c`. The executable is then named `filename.exe` and is run by typing `filename` at the DOS prompt.

2. Using gcc on a Unix Machine

When using EMACS, type `ESC-x compile` after entering your code in the editor. In the minibuffer, enter the command

```
gcc filename.c -lm
```

(With older versions of *gcc* you may have include a switch as in `gcc -traditional filename.c -lm.`) A small window will open which lists compilation errors, if any. If errors are detected, enter the command `ESC-x next-error`. This positions you in the source code at the offending point, and you can take appropriate action. Once the compilation succeeds, exit to the command line. The executable has been called `a.out` and you run it by simply typing `a.out`.

Should the program fail and dump core, you need to use `dbx` as follows. First, recompile your program with the `-g` option. Then enter `dbx -r a.out core`. Several (perhaps undecipherable) messages flash by, along with some hex addresses. When it pauses, enter `where`. The explicit line(s) will then be tagged, and you can hope to fix things.

In the above, `-lm` links to the math library, and `-traditional` (if needed) allows *gcc* to understand ANSI. The latest version of *gcc* (v. 2.0) compiles ANSI *C* by default. Some other useful options for gcc are

```
-o outfilename      renames the executable to ''outfilename''
-O                                        optimize for speed
```

In EMACS you can of course edit the compile command in your `.emacs` file to read as above. I suggest that you also edit your `.cshrc` (c-shell run commands) by including a line like this making 'gc' an alias:

```
alias 'gc gcc \!* -lm'
```

Lastly is a caveat on the use of the Sun Microsystems *C* compiler. As of this writing (1991–92), Sun has not yet updated its compiler with a switch rendering it capable of understanding ANSI *C*. Therefore the programs in this book will not run under the Sun *C* compiler.

HOW TO OBTAIN THE PROGRAMS

All of the programs in this book are freely available on Internet. Here is the procedure for obtaining them:

Enter the command

```
ftp iu-math.math.indiana.edu
```
or
```
ftp 129.79.147.6 .
```

At the login prompt `login:`, enter `anonymous`. Any string will work for the password.

Change directories by entering the command

```
cd pub/glassey .
```

If you are using UNIX, enter at the `ftp` prompt `binary`. Then get the relevant file by entering

```
get csrcrtg.tar.Z .
```

Once you have obtained this file, you can uncompress it with the UNIX command

```
uncompress csrcrtg.tar.Z .
```

Then enter the UNIX command

```
tar -xvf csrcrtg.tar .
```

For convenience we also place in the same directory an uncompressed ASCII listing of the programs whose file name is `csrcrtg.txt`. After changing directories as above, enter at the `ftp` prompt

```
get csrcrtg.txt .
```

CHAPTER 1

STD TUTORIAL

FIRST PRINCIPLES

Please recall my philosophy that the only way to learn a language is to read a book (this one, or [KR], or another), to read the code of others, and then to sit down and write programs yourself. I assume that you are already familiar with (or are now studying) Numerical Analysis. Program line numbers are *not* used in *C*. Nevertheless, for reference purposes we display source code using line numbers. `The program listings themselves appear in typewriter font.`

We begin with

```
/* Comments are enclosed like this */
```

The first example program will use *Simpson's Rule* to approximate the value of the integral $I = \int_0^1 \exp(-x^2)\,dx$. As you know, Simpson's Rule looks like this:

$$\int_a^b f(x)\,dx = \frac{(b-a)}{6}\left[f(a) + 4f\left(\frac{a+b}{2}\right) + f(b)\right] + E, \tag{1.1}$$

where the *error* $E = \frac{-(b-a)^5 f^{(4)}(\xi)}{2880}$ for some point $\xi \in (a,b)$. Let's ignore the error term for now so that I is approximately equal to

$$\frac{[\exp(0) + 4\exp(-0.25) + \exp(-1)]}{6} \approx 0.74718. \tag{1.2}$$

We'll write a *C* program to compute this:

```
 1 /* Simpson1.c */
 2
 3 # include <stdio.h>
 4 # include <math.h>
 5
 6 main( )
 7 {
 8 float s;         /* floating point variable */
 9                  /* to hold the sum */
10 s=(exp(0.0)+4.0*exp(-0.25)+exp(-1.0))/6.0;
11 printf("Integral=%f\n",s);
12 }
```

There is a lot here! Firstly, every *C* program has a "central" body called "main." Notice the empty parentheses after "main." The body of the program is always enclosed in braces, { on line 7 and } on line 12. Each executable statement in a *C* program is terminated by a semicolon, as you see on lines 8, 10 and 11. Above the main part of the program on lines 3 and 4 you see two "include" files, whose appearance is heralded by the # symbol. The file `math.h` contains type definitions of standard mathematical functions such as $\exp(x), \sin(x)$, etc. The file `stdio.h` is a similar "header" file where the compiler finds the required information on standard input and output. The program itself is simple. You must first define your variables in *C*. Here we have only one called s, which is a floating point number, hence the declaration `float s` on line 8. On line 10 s is computed according to the formula (1.2). It remains only to communicate the results, which one does by calling the function `printf`. (This stands for "print with formatting".) The line

```
printf("Integral=%f\n",s);
```

says: "print to the screen the value of the floating point variable `s`, calling it 'Integral' ". Any desired string output (such as the word "Integral" here) is enclosed in quotes. The symbol `%f` is a float format specifier; the "\n" gives us a new line in the output. Please note the use of real floating point values: 1.0 instead of 1, etc.

Let's change a few things, but only one at a time. Suppose the function you are integrating is more complicated: $f(x) = \exp(\sqrt{1+x^2} - x)$. We do not want to type this in three (or more) times, so we add at the top a *function definition*, or *function macro*, like this:

```
# define f(x)  (exp(sqrt(1.0+(x)*(x))-(x)))
```

There is a space after $f(x)$ and before the "(", but there must be no space between the "f" and the "(". Notice that the line itself is *not* terminated with a semicolon, and that the entire expression is enclosed within parentheses. The other relevant point is that each time the argument x appears, it is enclosed in its own set of parentheses. This is important; such a function-type macro will not execute correctly without these. Use parentheses liberally!

The entire program to compute $I = \int_0^1 f(x)\,dx$ now appears as:

```
/* Simpson2.c */

# include <stdio.h>
# include <math.h>
# define f(x) (exp(sqrt(1.0+(x)*(x))-(x)))

main( )
{
float s;    /* floating point variable */
            /* to hold the sum */
s=(f(0.0)+4.0*f(0.5)+f(1.0))/6.0;
printf("Integral=%f\n",s);
}
```

It's even easier, isn't it? What if the function $f(x)$ were much more complicated? Then we'd write a separate *C function* to find its value, and rewrite the Simpson program so that it receives this C-function as an argument (this is called a *pointer to a function*). This will be discussed in Chapter 2.

Perhaps you now think that the algorithm is too crude, which is true. Such an approximation over a larger interval is not likely to be very accurate. Therefore we now consider the *Composite Simpson's Rule*. The integral to be approximated is again written as $I = \int_a^b f(x)\,dx$. One partitions $[a, b]$ into equal subintervals

$$x_i = a + ih, \quad i = 0, 1, \ldots, N, \quad \text{where} \quad h = \frac{b-a}{N}. \tag{1.3}$$

We can write the composite rule as

$$\begin{aligned} I &= \frac{h}{6}\sum_{i=1}^{N}\Big(f(x_{i-1}) + 4f(x_{i-\frac{1}{2}}) + f(x_i)\Big) - E \\ &= \frac{h}{6}\left[f(a) + f(b) + 2\sum_{i=1}^{N-1} f(x_i) + 4\sum_{i=1}^{N} f(x_{i-\frac{1}{2}})\right] - E, \end{aligned} \tag{1.4}$$

where, for some point $\xi \in (a,b)$, the *error* E is given by

$$E = \frac{1}{2880}(b-a)f^{(4)}(\xi)h^4. \tag{1.5}$$

This form of the error does not result directly, but comes from the following "mean-value" theorem: Given a continuous function f on an interval $[a,b]$ and a sequence of values $\{g_i\}_{i=1}^n$ all of one sign, there exists a point $\xi \in [a,b]$ such that

$$\sum_{i=1}^n f(x_i)g_i = f(\xi)\sum_{i=1}^n g_i.$$

This is quite important and merits a few lines of proof. Without loss of generality we can assume that $g_i \geq 0$ for all i. Let the function $f(x)$ take values in the interval $[m,M]$ for $x \in [a,b]$. Then

$$m\sum_{i=1}^n g_i \leq \sum_{i=1}^n f(x_i)g_i \leq M\sum_{i=1}^n g_i, \quad \text{i.e.,}$$

$$m \leq \frac{\sum_{i=1}^n f(x_i)g_i}{\sum_{i=1}^n g_i} \leq M.$$

This says that the number

$$\frac{\sum_{i=1}^n f(x_i)g_i}{\sum_{i=1}^n g_i}$$

lies between the extreme values of f, and thus, by the intermediate value theorem, is equal to $f(\xi)$ for some point ξ.

To code this we evidently need additionally two integers (`int`), N and i. Then we simply add up (over $1 \leq i \leq N$) contributions such as those in the previous program using a *for loop*. Here it is for $N = 10$ and $I(f) = \int_0^1 \exp(\sqrt{1+x^2} - x)\,dx$:

```
/* Simpson3.c */

# include <stdio.h>
# include <math.h>
# define f(x) (exp(sqrt(1.0+(x)*(x))-(x)))

main( )
{
int i;
float h,s;   /* step size and sum variable */

12 s=f(0.0)+f(1.0);     /* initialize the sum */
13 h=1.0/10.0;      /* use 10 points this time */
14
15 for (i=1;i<=9;i++){
16         s = s + 4.0*f((i-0.5)*h)+2.0*f(i*h);
17         }              /* end the for loop */
18
19         s = s + 4.0*f(9.5*h);
20         s = s * h/6.0;
21
22 printf("Integral=%f\n",s);
23 }                              /* end main */
```

The notation $i++$ means increment i by one; similarly $i--$ means to decrement i by one. (There are more sophisticated properties of these operators, to be discussed later.) Notice the three parts in the *for* loop on line 15 are separated by semicolons, and that the entire loop is enclosed in braces. (While this is not necessary for a body consisting of only one line, it is not a bad practice to always enclose the loop body in braces.) The only other change is the integer i defined on line 9. The variable s is then initialized on line 12. *C* allows one to initialize variables at the time of their definition, so we could shorten the program by replacing the lines

```
float h,s;                      /* step size and sum variable */
s=f(0.0)+f(1.0);                        /* initialize the sum */
h=1.0/10.0;                        /* use 10 points this time */
```

by

```
float h=1.0/10.0,s=f(0.0)+f(1.0);
/* step size and sum variable */ .
```

There is some "type-mixing" above, e.g., what is the meaning of the expression $i * h$? We postpone this briefly; please see the discussion below.

C allows a shorthand to abbreviate operations such as $s = s + a$, $s = s * b$, etc. Thus we can write $s = s + a$ as $s += a$, and $s = s * a$ as $s *= a$, etc. This is very convenient and may be more efficient; you'll get used to it soon. Thus the sum computation in the *for* loop above (on line 16) could be succinctly written as

```
s += (4.0*f((i-0.5)*h)+2.0*f(i*h));
```

By the properties in Chapter 3, an expression of the form `x *= u+1.0` means `x = x*(u+1.0)`.

Of course we can write `for` loops which run "backward"

```
for (i=N;i>=1;i--)
```