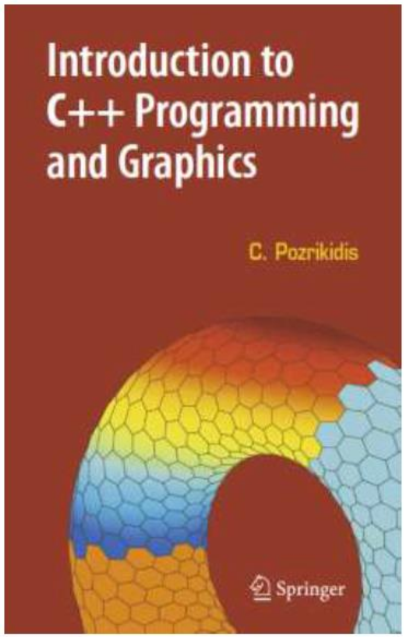
Programming and numerical analysis -conditional branch, iteration-

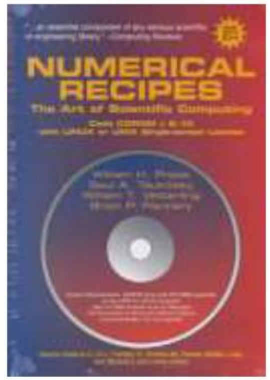
Hiroshi Sagara

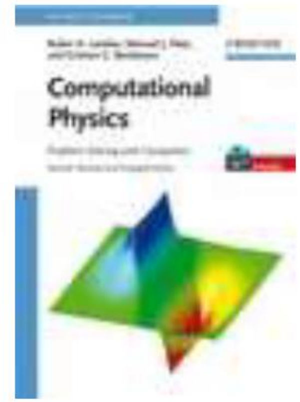
Oct. 13, 2020
Tokyo Tech

Tokyo Tech

Text Book







Numerical recipes in C++ : the art of scientific computing / William H. Press ... [et al.], ISBN:
9780521750332 [0521750334]

Introduction to C++ Programming and Graphics / Constantine Pozrikidis, Springer eBooks Computer Science, ISBN:
9780387689920 [0387689923],
https://link.springer.com/10.1007/978-0-387-68993-7

Computational Physics - Problem Solving with Computers / Rubin H. Landau, Cristian C. Bordeianu, Manuel José Páez Mejía
Wiley Online Library Online Books ISBN: 9783527406265 [3527406263] 9783527618835 [352761883X],
https://onlinelibrary.wiley.com/book/10.1002/9783527618835

Remind 2.7 Compiling in Unix

Text p.41 ~

Suppose that a self-contained C++ program has been written in a single file named *addition.cc*. To compile the program on a Unix system, we navigate to the directory where this file resides, and issue the command:

```
c++ addition.cc
```

This statement invokes the C++ compiler with a single argument equal to the file name. The compiler will run and produce an executable binary file named *a.out*, which may then be loaded into memory (executed) by issuing the command:

```
a.out
```

It is assumed that the search path for executables includes the current working directory where the *a.out* file resides, designated by a dot (.). To be safe, we issue the command:

```
./a.out
```

which specifies that the executable is in the current directory.

2.7 Compiling in Unix

Alternatively, we may compile the file by issuing the command:

```
c++ -o add addition.cc
```

This will produce an executable file named *add*, which may then be loaded (executed) by issuing the command:

```
add
```

or the safer command:

```
./add
```

Other compilation options are available, as explained in the compiler manual invoked by typing:

```
man gcc
```

for the GNU project C and C++ compilers.

3 Programming in C++

Having illustrated the general structure of a C++ program, we now turn to discussing the basic operators, commands, and logical constructs. Most of these are either identical or similar to those encountered in other languages. However, C++ supports some unconventional and occasionally bizarre operations that require familiarization.

In Appendix C, a correspondence is made between MATLAB, FORTRAN 77, and C++ in the form of a dictionary that explains how to translate corresponding code.

3.1 Operators

Operators apply to one variable or a group of variables to carry out arithmetic and logical tasks.

Assignation

The equal sign (=) is the assignation or right-to-left copy operator. Thus, the statement

```
a = b;
```

means "replace the value of a with the value of b", and the statement

```
a = a+5;
```

means "replace the value of a with itself augmented by 5".

The assignation operator is distinguished by lack of reciprocity: the statement `a=b` is different from the statement `b=a`.

3.1 Operators

Arithmetic operators

The basic implementation of C++ supports the following arithmetic operators:

- Addition (+): We may write

```
c=a+b;
```

- Subtraction (-): We may write

```
c=a-b;
```

- Multiplication (*): We may write

```
c=a*b;
```

- Division (/): We may write

```
c=a/b;
```

- Modulo (%): We may write

```
c=a%b;
```

  This operator extracts the remainder of the division a/b. For example

$$5\%3 = 2$$

3.1 Operators

Unconventional operators

In C++, we can write:

```
a = b = c = 0.1;
```

with the expected result. A perfectly acceptable C++ statement is:

```
a = 1 + (b = 3);
```

meaning:

```
b=3;
a = 1 + b;
```

3.1 Operators

Compound assignation

Other unconventional statements mediated by compound assignation operators are listed in Table 3.1.1.

Operation	*Meaning*
a +=b;	a=a+b;
a -=b;	a=a-b;
a *=b;	a=a*b;
a /=b;	a=a/b;
a *= b+c;	a=a*(b+c);
a++;	a=a+1;
++a;	a=a+1;
a--;	a=a-1;
--a;	a=a-1;

Table 3.1.1 Unconventional statements mediated by compound assignation operators in C++. The language name C++ translates into C+1, which subtly indicates that C++ is one level above C. Alternatively, we could have given to C++ the name C and rename C as C−−.

3.1 Operators

To illustrate the difference between the `a++` and `++a` operators, we issue the commands:

```
a = 5;
b = a++;
```

After execution, `a=6` and `b=5`.

Alternatively, we issue the commands:

```
a = 5;
b = ++a;
```

After execution, `a=6` and `b=6`.

3.1 Operators

Relational and logical operands

Relational and logical operands are shown in Table 3.1.2. For example, to find the maximum of numbers `a` and `b`, we write:

```
max = (a>b) ?  a :  b;
```

If $a > b$ is true, the variable max will set equal to a; if $a > b$ is false, the variable max will set equal to b.

Equal to	a == b
Not equal to	a != b
Less than	a<b
Less than or equal to	a<=b
Greater than	a>b
Greater than or equal to	a>=b
And	A && B
Or	A || B
Boolean opposite or true or false	!A
Conditional operator	A ? a : b;

Table 3.1.2 Relational and logical operands in C++; a, b are variables, and A, B are expressions. The conditional operator shown in the last entry returns the value of the variable a if the statement `A` is true, and the value of the variable b if the statement `A` is false.

3.1 Operators

Threading

The statement:

```
c = (a=1, b=2, a+b);
```

is a compact representation of the statements:

```
a=1;
b=2;
c=a+b;
```

In these constructions, the variable c is evaluated from the rightmost expression inside the parentheses.

3.2 Vector and matrix initialization

To declare and initialize a vector v whose three elements are real numbers registered in double precision, we write

```
double v[3] = {1.0, 2.0, 4.5};
```

or

```
double v[] = {1.0, 2.0, 4.5};
```

which sets: $v[0] = 1.0$, $v[1] = 2.0$, $v[2] = 4.5$.

If we declare and initialize:

```
double v[5] = {1.0, 2.0};
```

then: $v[0] = 1.0$, $v[1] = 2.0$, $v[2] = 0.0$, $v[3] = 0.0$, $v[4] = 0.0$. Thus, the uninitialized values of a partially initialized vector are set to zero.

3.2 Vector and matrix initialization

If we only declare and not initialize by stating:

```
double v[5];
```

then the vector components are undefined.

Declaration and initialization must be done in a single line. We may not first declare and then initialize a vector.

Similarly, we can write

```
char u[3]= {78, 34, 78};
```

```
char e[10]= {'a', 'b', 'c'};
```

```
char q[]= 'zei';
```

```
string n[3]= {"who", "am", "I?"};
```

```
string b[]= {"who", "are", "they?"};
```

The size of `q` is four, as a final 0 is appended to indicate the end of a character array.

3.2 Vector and matrix initialization

To declare and initialize a 2×2 matrix A whose elements are real numbers registered in double precision, we write

```
double A[2][2] = { {1.0, 2.0}, {4.5,-3.5} };
```

or

```
double A[][] = { {1.0, 2.0}, {4.5,-3.5} };
```

which sets: $A[0][0] = 1.0$, $A[0][1] = 2.0$, $A[1][0] = 4.5$, $A[1][1] = -3.5$.

Thus, the matrix elements are initialized row-by-row.

Similarly, we can write

```
char D[2][3]= { {60, 61, 65}, {62, 63, 66} };

string C[2][3]= { {"who", "am", "I?"}, {"who", "is", "she?"} };

string C[][]= { {"who", "am", "I?"}, {"who", "is", "she?"} };
```

3.3 Control structures

Text p.53 ~

Control structures are blocks of statements that implement short algorithms and make logical decisions based on available options. An algorithm is a set of instructions that achieves a goal through sequential or repetitive steps.

C++ employs control structures with single or multiple statements. The former are simply stated, while the latter are enclosed by curly bracket delimiters, {}.

3.3 Control structures

- if statement:

  The `if` statement implements conditional execution of one command or a block of commands.

  For example, we may write

```
if(a==10)
  b=10;
```

  or

```
if(a==10)
  {
  b=10;
  }
```

  If more than one statements is involved, the use of curly brackets is mandatory:

```
if(a!=10)
  {
  b=a+3;
  c=20;
  }
```

  We highly recommend using the curly brackets even in the case of one statement.

3.3 Control structures

- if/else structure:

The `if/else` structure implements conditional execution based on two options.

For example, we may write:

```
if(a!=10)
  {
```

```
  b=a+3;
  c=20;
  }
else
  {
  cout << "angouraki" << endl;
  }
```

The statement

```
cout << "angouraki" << endl;
```

prints the word “angouraki” on the screen and moves the cursor to the next line.

3.3 Control structures

- if/else if structure:

The `if/else if` structure implements conditional execution based on several options.

For example, we may write:

```
if(a==1)
  {
  b=a+3;
  c=20;
  }
else if (a==2.3)
  {
  cout << "angouraki" << endl;
  }
else
  {
  cout << "maintanos" << endl;
  }
```

We can use multiple `else if` blocks and skip the last `else` block. If two options coincide, the first-encountered option will be executed before exiting the structure.

3.3 Control structures

- switch structure:

Consider an integer or character variable, *diosmos*. If $diosmos = n1$ we want to execute a block of commands, if $diosmos = n2$ we want to execute another block of commands, if $diosmos = n3$ we want to execute a third block of commands; otherwise, we want to execute a default block of commands.

These conditional choices are best implemented with the switch structure:

```
switch(diosmos)
case n1:
  {
  ...
  }
  break;
case n2:
  {
  ...
  }
  break;
...
default:
  {
  ...
  }
```

The default block at the end is not mandatory. Note that this block does not contain a `break;` .

3.3 Control structures

- for loop:

To compute the sum: $s = \sum_{i=1}^{N} i$, we use the `for` loop:

```
double s=0;
int i;

for (i=1; i<=N; i+1)
  {
  s = s + i;
  }
```

The plan is to first initialize the sum to zero, and then add successive values of i. The `i+1` expression in the argument of the `for` statement can be written as `i++`.

3.3 Control structures

- Break from a for loop:

  To escape a `for` loop, we use the command `break`.

For example, to truncate the above sum at $i = 10$, we use:

```
double s=0;

for (int i=1; i<=N; i++)
  {
  if(i==10) break;
  s = s + i;
  }
```

3.3 Control structures

- Skip a cycle in a for loop:

  To skip a value of the running index in a `for` loop, we use the command `continue`.

  For example, to skip the value $i = 8$ and continue with $i = 9$ and 10, we use:

```
double s=0;

for (int i=1; i<=10; i++)
  {
  if(i==8) continue;
  s = s + i;
  }
```

3.3 Control structures

- goto:

  We use this statement to jump to a desired position in the code marked by a label designated by a colon (:).

For example, consider the block of commands:

```
goto mark;
a=5;
mark:
```

The statement `a=5` will be skipped.

FORTRAN 77 users are fondly familiar with the `Go to` statement. MATLAB users are unfairly deprived of this statement.

3.3 Control structures

- while loop:

  We use the `while` loop to execute a block of commands only when a distinguishing condition is true.

  For example, the following `while` loop prints the integers: 1, 2, ..., 9, 10:

```
int i=0;

while(i<10)
{
i=i+1;
cout << i << " ";
}
```

  Note that the veracity of the distinguishing condition `i<10` is checked *before* executing the loop enclosed by the curly brackets.

3.3 Control structures

- do-while:

  This is identical to the `while` loop, except that the veracity of the distinguishing condition is examined *after* the first execution of the statements enclosed by the curly brackets. Thus, at least one execution is granted even if the distinguishing condition is never true.

  For example, the `do-while` loop

  ```
  int i=0;

  do
    {
    i=i+1;
    cout << i << " ";
    }
    while(i<10);
  ```

  prints the integers: 1, 2, 3, ..., 9, 10.

  The `do-while` loop is favored when a variable in the distinguishing condition is evaluated inside the loop itself, as in our example.

3.3 Control structures

- exit:

To stop the execution at any point, we issue the command:

```
exit(1);
```

The use of these control structures will be exemplified throughout this book.

Text p.59 ~

The `iostream` library allows us to enter data from the keyboard and display data on the monitor. In computer science, the keyboard is the standard input and the monitor is the standard output.

It is illuminating to view the keyboard and monitor as abstract objects that can be replaced by files, printers, and other hardware or software devices. The mapping of physical to abstract objects is done by software interfaces called device drivers.

To read a numerical variable from the keyboard, we issue the statement:

```
cin >> variable;
```

On execution, the computer will wait for input followed by the ENTER key.

To read two numerical variables, we use either the separate statements:

```
cin >> variable1;
cin >> variable2;
```

or the composite statement:

```
cin >> variable1 >> variable2;
```

On execution, the computer will wait for two inputs separated by a space, comma, or the ENTER keystroke.

Leading white space generated by the space bar, the tab key, and the carriage return is ignored by the `cin` function.

Now consider the following block of commands:

```
double pi;
int a;
cin >> pi;
cin >> a;
```

Suppose that, on execution, we enter the number π in segments separated by white space:

```
3.14159 265358
```

Since the two `cin` statements are equivalent to:

```
cin >> pi >> a;
```

the program will take

`pi=3.14159` `a=265358`.

Thus, the computer will not pause for the second `cin`, giving the false impression of a coding error.

In professional codes, we circumvent this difficulty by reading all input date as strings, and then making appropriate data type conversions.

Displaying on the monitor

To display the value of a numerical variable on the monitor, we issue the command:

```
cout << variable;
```

To display the value of a numerical variable and move the cursor to the next line, we use:

```
cout << variable << "\n";
```

To print a message on the screen and move the cursor to the next line, we use:

```
cout << "hello\n";
```

To display the values of two numerical variables separated by space and move the cursor to the next line, we use:

```
cout << variable << " " << variable1 << " total" << endl;
```

Material enclosed by double quotes is interpreted verbatim as text. The text directive "\n", and its equivalent end-of-line directive "endl", both instruct the cursor to move to the next line.

Other printing codes preceded by the backslash are shown in Table 3.4.1. For example, we can sound a beep by printing: \a.

\'	Print a single quote (')
\"	Print a double quote (")
\?	Print a question mark (?)
\\	Print a backslash (\)
\a	Sound a beep
\t	Press the tab key
\v	Issue a vertical tab
\r	Issue a carriage return
\b	Issue a backspace signal
\f	Issue a page feed
\n	Issue a line break
\\	Continue a string to the next line

Table 3.4.1 Printing codes preceded by the backslash.

Text p.68 ~

Table 3.5.1 lists functions of the C++ mathematical library. To use these functions, the associated header file must be included at the beginning of the program by stating:

```
#include <cmath>
```

For example, to compute the exponential of a number a, we write:

```
#include <cmath>

float a = 2.3;
float b = exp(a);
```

Equally well, we can write

```
double b = exp(2.3);
```

$m = \mathrm{abs}(n)$	Absolute value of an integer, n
$y = \mathrm{acos}(x)$	Arc cosine, $0 \le y \le \pi$
$y = \mathrm{asin}(x)$	Arc sine, $-\pi/2 \le y \le \pi/2$
$y = \mathrm{atan}(x)$	Arc tangent, $-\pi/2 \le y \le \pi/2$
$y = \mathrm{atan2}(x, z)$	Arc tangent, $y = \mathrm{atan}(y/z)$
$y = \mathrm{ceil}(x)$	Ceiling of x (smallest integer larger than or equal to x)
$y = \cos(x)$	Cosine
$y = \cosh(x)$	Hyperbolic cosine
$y = \exp(x)$	Exponential
$y = \mathrm{fabs}(x)$	Absolute value of a real number, x
$y = \mathrm{floor}(x)$	Floor of x (smallest integer smaller than or equal to x)
$y = \log(x)$	Natural log
$y = \mathrm{log10}(x)$	Base-ten log
$y = \mathrm{pow}(x, a)$	$z = x^a$, where x and a are real
$y = \sin(x)$	Sine
$y = \sinh(x)$	Hyperbolic sine
$y = sqrt(x)$	Square root
$y = \tan(x)$	Tangent
$y = \tanh(x)$	Hyperbolic tangent

Table 3.5.1 Common C++ mathematical functions. The statement `#include <cmath>` must be included at the preamble of the program.

3.5 Mathematical library

The argument and return of the mathematical functions are registered in double precision (double). If an argument is in single precision (float), it is automatically converted to double precision, but only for the purpose of function evaluation.

Text p.70 ~

We have learned how to read data from the keyboard and write data to the screen. To read data from a file and write data to a file, we use the intrinsic library `fstream`.

Read from a file

To read from a file named *stresses.dat*, we simply associate the file with a *device* that replaces `cin` of the `iostream`:

```
#include<fstream>
ifstream dev1;
dev1.open("stresses.dat");
dev1 >> variable1 >> variable2;
dev1.close();
```

The first line declares the device *dev1* as a member of the “input file stream.” The second line opens the file through the device, the third line writes to the device, and the fourth line closes the device.

Note that *device* and *filename* are two distinct concepts. A brilliant notion of C++ (and Unix) is that we can change the device but keep the filename.

In compact notation, the lines

```
ifstream dev1;
dev1.open("stresses.dat");
```

can be consolidated into one,

```
ifstream dev1("stresses.dat");
```

which bypasses the explicit use of the `open` statement.

Suppose that we want to read the components of a vector from a file, but the length of the vector is unknown so that we cannot use a `for` loop. Our best option is to use a `while` loop based on a false read.

3.6 Read from a file and write to a file

The implementation of the algorithm is:

```
#include <iostream>
#include <fstream>
using namespace std;

int main()
{
ifstream file9("vector.dat");
int i=1;
double a[10];

while(file9 >> a[i])
{
cout << i << " " << a[i] << endl;
i++;
}

file9.close();
return 0;
}
```

If the file *vector.dat* reads:

```
3.4 9.8
3.0 9.1
0.45
```

the output of the code will be:

```
1 3.4
2 9.8
3 3
4 9.1
5 0.45
```

A false read arises when either the program has reached the end of a file (EOF), or the program attempts to read a certain data type and sees another.

Write to a file

To write to a file named *post_process.dat*, we simply associate the file with a device that replaces `cout` of the `iostream`:

```
#include<fstream>

ofstream dev2;
dev2.open("post_process.dat");
dev2 << variable1 << variable2;
dev2 << variable << " " << variable1 << " total" << endl;
dev2.close();
```

The second line declares the device *dev2* as a member of the "output file stream." The third line opens the device, the fourth line writes to the device, and the fifth line closes the device.

The second and third statements can be consolidated into one,

```
ofstream dev2("post_process.dat");
```

which bypasses the explicit use of the `open` statement.

Parameter	*Meaning*
in	Input mode (default for a file of the `ifstream` class)
out	Output mode (default for a file of the `ofstream` class)
binary	Binary mode
app	If the file exists, data is written at the end (appended)
ate	For a new file, data is written at the end For an existing file, data is written at the current position (same as app but we can write anywhere)
trunc	If the file exists, delete the old content (same as out)
noreplace	If the file exists, do not open
nocreate	If the file does not exist, do not open

Table 3.6.1 Open-file parameters for reading data from a file and writing data to a file.

Text p.74 ~

The input/output manipulation library `iomanip` allows us to print data in an orderly fashion. As an example, consider the program:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
double pi;
pi=3.14159265358;
cout << setprecision(5) << setw(10);
cout << pi << endl;
return 0;
}
```

Running the program prints on the screen:

```
    3.1416
```

In this case, the set-width manipulator `setw(10)` reserves ten spaces, and the set-precision manipulator `setprecision(5)` allocates five of these spaces to the decimal part, including the decimal point.

The code:

```
for (int i=1;i<3;i++)
{
  for (int j=1;j<5;j++)
  {
  cout <<"+"<< setfill('-')<<setw(4);
  }
  cout<< "+" << endl;
}
```

prints on the screen the pattern:

```
+---+---+---+---+
+---+---+---+---+
+---+---+---+---+
```

Tokyo Tech

Table 3.7.1 presents I/O manipulators with brief descriptions. Some of these manipulators apply to only one read or write, whereas others apply permanently until reset.

Manipulator	*Manipulator*	*Comment*
setw(n)	width(n)	Set the minimum field width
setprecision(n)		Set the number of digits printed to the right of the decimal point
showpoint	noshowpoint	Decimal point
uppercase	nouppercase	
dec	oct	Decimal or octal form
hex	setbase(8—10—16)	Hexadecimal
left	right	Margin justification used after setw(n)
showbase	noshowbase	
setfill(ch)		Fill empty fields with a character
boolalph	anoboolalpha	Boolean format
fixed	scientific	Notation
ends		
showpos	noshowpos	
skipws	noskipws	Skip white space in reading
ws		Ignore white space at the current position
internal	flush	
unitbuf	nounitbuf	
setiosflags(f)	resetiosflags(f)	

Table 3.7.1 Input/Output manipulators for formatted reading and printing.

Tabulation

The following code contained in the file *tabulate.cc* prints a table of exponentials:

```
#include <iostream>
#include <iomanip>
#include <cmath>
using namespace std;

int main()
{
  int i;
  double step=0.1;
  cout << setiosflags(ios::fixed | ios::showpoint);

  for (i=1;i<=6;i++)
  {
  double x=(i-1.0)*step;
```

```
    double y=exp(x);
    cout << setprecision(2) << setw(5) << x << " ";
    cout << setprecision(5) << setw(7) << y << endl;
    }

  return 0;
  }
```

The output of the code is:

```
0.00 1.00000
0.10 1.10517
0.20 1.22140
0.30 1.34986
0.40 1.49182
0.50 1.64872
```

What would the output be if the `setiosflags()` manipulator were not included?

Random numbers

As a second application, we discuss a code contained in the file *random.cc* that computes and prints on the screen random numbers with uniform probability distribution in the range [0, 1], also called uniform deviates, using the C++ compiler random-number generator:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
int N=6, random_integer;

float random_real, random_number, max=RAND_MAX;

cout<< setiosflags(ios::fixed | ios::showpoint);

for(int i=1;i<=N;i++)
{
random_integer = rand();
random_real = random_integer;
random_number = random_real/max;
cout << setw(3) << i << " " << setw(6) << setprecision(5)
          << random_number << endl;
}

return 0;
}
```

The internal C++ function `rand` generates random integers ranging from 0 up to the maximum value of `RAND_MAX`. Converting these integers to real numbers and normalizing by the maximum generates the requisite list. The output of the code is:

```
1 0.84019
2 0.39438
3 0.78310
4 0.79844
5 0.91165
6 0.98981
```

Tokyo Tech

Text p.80 ~

We have learned how to enter data from the keyboard, print data to the screen, read data from a file, and write data to a file. In scientific and other applications, the data are manipulated according to carefully designed algorithms to achieve a specific goal.

We have defined an algorithm as a set of instructions that achieves a goal through sequential or repetitive steps. Certain algorithms provide us with systematic ways of eliminating events and narrowing down possibilities. Other algorithms provide us with craftily devised methods of producing a sequence of approximations to a desired solution.

Indexing an array

Now we want to index the elements of the array `a[l]`, for $l = 1, \ldots, N$, so that the index of the largest number is equal to 1, and the index of the smallest number is equal to N. The following code contained in the file *index.cc* uses the ranking algorithm to index the array:

```
#include <iostream>
#include <iomanip>
using namespace std;

int main()
{
float a[6]={0.0, 8.0, 9.7, -1.4, -8.0, 13.8};
int i,j;
int m[6];    // indexing array
const int N=5;

for(i=1; i<=N; i++)
{
  m[i]=1;
  for(j=1; j<=N; j++)
  {
  if(a[i]<a[j] && i!=j) m[i]++;
  }
}

//--- print the list

cout << fixed << showpoint;
for(i=1; i<=N; i++)
cout << setw(8) << setprecision(2) << a[i] << " " << m[i] << endl;

}
```

The output of the code is:

```
  8.00 3
  9.70 2
 -1.40 4
 -8.00 5
 13.80 1
```

Bubble sort

It is often necessary to sort an array of numbers contained in a vector `x[i]`. The sorting can be done in ascending order where the largest number is placed at the bottom, or in descending order where the smallest number is placed at the bottom.

In the *bubble-sort* algorithm, we first find the highest number, and put it at the bottom of the list. This is done by comparing the first number with the second number and swapping positions if necessary, then comparing the second with the third number and swapping positions if necessary, and repeating the comparisons all the way to the bottom. In the second step, we find the second-largest number and put it in the penultimate position using a similar method. In this fashion, light numbers "bubble up" to the top. The algorithm is implemented in the following code contained in the file *bsort.cc*:

3.8 Sample algorithms

```
#include<iostream>
#include<iomanip>
using namespace std;

int main()
{
const int n=5;
float save, x[n+1]={0.0, -0.5, -0.9, 0.3, 1.9, -0.3 };
int Istop, i, k;

//--- bubble sort:

k = n-1; // number of comparisons

do {
  Istop = 1; // will stop if Iflag 1
  for (i=1;i<=k;i++) // compare
  {
  if(x[i]>x[i+1])
    {save = x[i]; // swap
    x[i]=x[i+1];
    x[i+1] = save;
    Istop = 0; // an exchange occurred; do not stop
    }
  }
  k--; // reduce the number of comparisons
}
while(Istop==0);
```

```
//--- print the sorted array:

for (i=1;i<=n;i++)
{
cout << setw(5) << right << x[i] << endl;
};

return 0;
}
```

The output of the code is:

```
-0.9
-0.5
-0.3
 0.3
 1.9
```

Problems

3.8.1. Modify the bubble-sort code to arrange the array in descending order with the smallest number put at the bottom.