# C++ and Object Oriented Numeric Computing for Scientists and Engineers

Daoqi Yang

*To Xiujuan, Joy, and Forrest*

# Preface

This book is intended to be an easy, concise, but rather complete, introduction to the ISO/ANSI C++ programming language with special emphasis on object-oriented numeric computation for students and professionals in science and engineering. The description of the language is platform-independent. Thus it applies to different operating systems such as UNIX, Linux, MacOS, Windows, and DOS, as long as a standard C++ compiler is equipped. The prerequisite of this book is elementary knowledge of calculus and linear algebra. However, this prerequisite is hardly necessary if this book is going to be used as a textbook for teaching C++ and all the sections on numeric methods are skipped. Programming experience in another language such as FORTRAN, C, Ada, Pascal, Maple, or Matlab will certainly help, but is not presumed.

All C++ features are introduced in an easy way through concepts such as functions, complex numbers, vectors, matrices, and integrals, which are familiar to every student and professional in science and engineering. In the final chapter, advanced features that are not found in FORTRAN, C, Ada, or Matlab, are illustrated in the context of iterative algorithms for linear systems such as the preconditioned conjugate gradient (CG) method and generalized minimum residual (GMRES) method. Knowledge of CG, GMRES, and preconditioning techniques is not presumed and they are explained in detail at the algorithmic level. Matrices can be stored in full (all entries are stored), band (entries outside a band along the main diagonal are zero and not stored to save memory), and sparse (only nonzero entires are stored to save memory) formats and exactly one definition for CG or GMRES is needed that is good for all three matrix storage formats. This is in contrast to a procedural language such as FORTRAN, C, Ada, or Matlab, where three definitions of CG and GMRES may have to be given for the three matrix formats. This is one of the salient features of object-oriented programming called *inheritance*. The CG and GMRES methods are defined for a base *class* and can be inherited by classes for full, band, and sparse matrices. If one later decides to add, for example, a symmetric (full, band, or sparse) storage format (only half of the entries need be stored for a symmetric matrix to save memory), the code of CG and GMRES methods can be reused for it without any change or recompilation.

Another notable feature is generic programming through *template*s, which enables one to define a function that may take arguments of different data types at different invocations. For example, the same definition of CG and GMRES can deal with matrices with entries in single, double, and extended double (*long double*) precisions and complex numbers with different types for real and imaginary parts. Again, using C or FORTRAN would require different versions of the same function to handle different data types; when the implementation of the function is going to be changed later for robustness or efficiency, then every version has to be changed, which is tedious and error-prone. The *operator overloading* feature of C++ enables one to add and multiply matrices and vectors using $+$ and $*$ in the same way as adding and multiplying numbers. The meaning of the operators is user-defined and thus it provides more flexibility than Matlab. For example, the operators $+$ and $*$ can be used to add and multiply full, band, and sparse matrices. These and other features of C++ such as information hiding, encapsulation, polymorphism, error handling, and standard libraries are explained in detail in the text. With increasingly complicated numeric algorithms, many sophisticated data structures can be relatively easily implemented in C++, rather than in FORTRAN or C; this is a very important feature of C++. The C++ compiler also checks for more type errors than FORTRAN, C, Ada, Pascal, and many other languages do, which makes C++ safer to use.

However, there are trade-offs. For example, *template*s impose compile-time overhead and *inheritance* (with dynamic type binding) may impose run-time overhead. These features could slow down a program at compile- and run-times. A good C++ compiler can minimize these overheads and a programmer can still enjoy programming in C++ without suffering noticeable compile- and run-time slowdowns but with spending possibly much less time in code development. This is evidenced by the popularity of C++ in industry for database, graphics, and telecommunication applications, where people also care dearly about the speed of a language. The fact is that C++ is getting faster as people are spending more time optimizing the compiler. Some performance comparisons on finite element analysis have shown that C++ is comparable to FORTRAN 77 and C in terms of speed. On the other hand, C++ has the potential of outperforming FORTRAN 77 and C for CPU-intensive numeric computations, due to its built-in arithmetic compound operators, template mechanisms that can, for example, avoid the overhead of passing functions as arguments to other functions, and high performance libraries (like *valarray*), which can be optimized on different computer architectures. This potential has become a reality in many test examples.

This book consists of three parts. The first part (Chapters 1 to 4) is an introduction to basic features of C++, which have equivalents in FORTRAN 90 or C. When a person knows only this part, he can do whatever a FORTRAN 90 or C programmer can do. A brief introduction is also made to *Makefile*, debuggers, making a library, and how to time and profile a program. The second part (Chapters 5 to 9) is on object-oriented and generic programming features of C++, which can not be found in FORTRAN 90 or C. Many techniques for writing efficient C++ programs are also included; see §6.5, §7.2, §7.6, §7.2.4, §7.7, and §8.6. It is this part that causes many people to think that C++ is "too complicated." To make it easy to understand for a typical science and engineering student or professional, I make use of basic concepts and operations for complex numbers, vectors, matrices, and integrals. This should be readily acceptable by a person with elementary knowledge of calculus and matrix algebra. The third part (Chapters 10 and 11) provides a description of C++ standard libraries on containers (such as linked list, set, vector, map, stack, and queue) and algorithms (such as sorting a sequence of elements and searching an element from a sequence according to different comparison criteria), and an introduction to a user-defined numeric linear algebra library (downloadable from my Web page) that contains functions for preconditioned conjugate gradient and generalized minimum residual methods for real and complex matrices stored in full, band, and sparse formats. Gauss elimination with and without pivoting is also included for full and band matrices. This enhances the reader's understanding of Parts 1 and 2. Furthermore, it can save the reader a great deal of time if she has to write her own basic numeric linear algebra library, which is used on a daily basis by many scientists and engineers. Great care has been taken so that features that are more easily understood and that are more often used in numeric computing are presented first. Some examples in the book are taken from or can be used in numeric computing libraries, while others are made up only to easily illustrate features of the C++ language. A Web page (http://www.math.wayne.eud/˜yang/book.htm) is devoted to the book on information such as errata and downloading programs in the book.

On the numeric side, discussions and programs are included for the following numeric methods in various chapters: polynomial evaluation (§3.12), numeric integration techniques (§3.13, §5.1, and §7.7), vector and matrix arithmetic (§6.3 and §7.1), iterative algorithms for solving nonlinear equations (§4.7), polynomial interpolation (§7.8), iterative and direct algorithms for solving systems of linear equations in real and complex domains (§6.6, §11.3, and §11.4), Euler and Runge-Kutta methods for solving ordinary differential equations (§5.9), and a finite difference method for solving partial differential equations (§11.5) with real and complex coefficients. The coverage of numeric algorithms is far from being complete. It is intended to provide a basic understanding of numeric methods and to see how C++ can be applied to program these methods efficiently and elegantly. Most of them are covered at the end of a chapter. People with an interest in learning how to program numeric methods may read them carefully, while readers who just want to learn C++ may wish to skip them or read them briefly.

C++ is not a perfect language, but it seems to have all the features and standard libraries of which a programmer can dream. My experience is that it is much easier to program than FORTRAN and C, because FORTRAN and C give too many run-time errors and have too few features and standard libraries. Many such run-time errors are hard to debug but can easily be caught by a C++ compiler.

After all, C++ is just a set of notation to most people and a person does not have to know all the features of C++ in order to write good and useful programs. Enjoy!

**How to Use This Book:**

This book can be used as a textbook or for self-study in a variety of ways.

1. The primary intent of this book is to teach C++ and numeric computing at the same time, for students and professionals in science and engineering. C++ is first introduced and then applied to code numeric algorithms.

2. It can also be used for people who just want to learn basic and advanced features of C++. In this case, sections on numeric computing can be omitted, and knowledge of calculus and linear algebra is not quite necessary. However, in some sections, the reader should know what a vector and a matrix are, and basic operations on vectors and matrices, such as vector addition, scalar-vector multiplication, and matrix-vector multiplication.

3. This book can be used as a reference for people who are learning numeric methods, since C++ programs of many numeric methods are included. Students who learn numeric methods for the first time often have difficulty programming the methods. The C++ code in the book should help them get started and have a deeper understanding of the numeric methods.

4. For experienced C++ programmers, this book may be used as a reference. It covers essentially all features of the ISO/ANSI C++ programming language and libraries. It also contains techniques for writing efficient C++ programs. For example, standard operator overloading for vector operations, as introduced in most C++ books, may be

a few times slower than using C or FORTRAN style programming. Techniques are given in the book so that the C++ code is no slower than its corresponding C or FORTRAN style code. Examples are also given to replace certain virtual functions by static polymorphism to improve run-time efficiency. Other advanced techniques include expression templates and template metaprograms.

**Acknowledgments:**

This book has been used as the textbook for a one-semester undergraduate course on C++ and numeric computing at Wayne State University. The author would like to thank his students for valuable suggestions and discussions. Thanks also go to the anonymous referees and editors whose careful review and suggestions have improved the quality of the book. Readers' comments and suggestions are welcome and can be sent to the author via email (dyang@na-net.ornl.gov).

Daoqi Yang
Wayne State University
Detroit, Michigan

June 28, 2000

# Contents

# 1
# Basic Types

This chapter starts with a sample C++ program and then presents basic data types for integral and floating point types, and two special types called *bool* and *void*. Towards the end of the chapter, numeric limits are introduced such as the largest integer and smallest double precision number in a particular C++ implementation on a particular computer. Finally, identifiers and keywords are discussed.

## 1.1 A Sample Program

A complete program in C++ must have a function called *main*, which contains statements between braces { and }. Each statement can extend to more than one line, but must end with a semicolon. Comments must be enclosed by /* and */, or preceded by // extending to the rest of the line. The first can be used for comments that stand on many lines while the second for a whole line or an end part of a line. One of them can be used to comment out the other. Comments and blank lines are ignored by the compiler and contribute only to the readability of a program.

Below is a simple program that calculates the sum of all integers between two given integers. The user is prompted to enter these two integers from the standard input stream (usually the terminal screen) represented by *cin*. The result is output to the standard output stream (usually the terminal screen) represented by *cout*. The streams *cin* and *cout* are defined in a C++ standard library called <*iostream*>. The program reads:

```
/* A sample program to illustrate some basic features of C++.
   It adds all integers between two given integers and
   outputs the sum to the screen.                             */

#include <iostream>      // include standard library for I/O
using namespace std;

main() {
  int n, m;              // declare n and m to be integers
  cout << "Enter two integers: \n";     // output to screen
  cin >> n >> m;         // input will be assigned to n, m

  if (n > m) {           // if n is bigger than m, swap them
    int temp = n;        // declare temp and initialize it
    n = m;               // assign value of m to n
    m = temp;            // assign value of temp to m
  }

  double sum = 0.0;      // sum has double precision

  // a loop, i changes from n to m with increment 1 each time
  for (int i = n; i <= m; i++) { // <=: less than or equal to
    sum += i;                    // sum += i: sum = sum + i;
  }
  cout << "Sum of integers from " << n << " to " << m
       << " is: "<< sum << '\n'; // output sum to screen
}
```

The first three lines in the program are comments enclosed by /* and */, which are usually used for multiline comments. Other comments in the program are short ones and preceded by two slashes //; they can start from the beginning or middle of a line extending to the rest of it.

Input and output are not part of C++ and their declarations are provided in a header file called *iostream* that must be included at the beginning of the program using

```
#include <iostream>
```

with the sign # standing in the first column. The statement

```
using namespace std;
```

lets the program gain access to declarations in the *namespace std*. All standard libraries are defined in a *namespace* called *std*. The mechanism *namespace* enables one to group related declarations and definitions together and supports modular programming; see Chapter 4 for more details. Before standard C++, these two statements could be replaced by including

*<iostream.h>*. The compiler will not recognize the identifiers *cin* and *cout* if the standard library *<iostream>* is not included. Similarly, the math library *<math.h>* must be included when mathematical functions like *sin* (sine), *cos* (cosine), *atan* (arctangent), *exp* (exponential), *sqrt* (square root), and *log* (logarithm) are used. The header files for standard libraries reside somewhere in the C++ system, and a programmer normally does not have to care where. When they are included in a program, the system will find them automatically. See §4.2 for more information on how to include a file.

The symbol $\ll$ is called the output operator and $\gg$ the input operator. The statement

```
cout << "Enter two integers: \n";
```

outputs the string "*Enter two integers:*" followed by a new line to the screen, telling the user to type in, on the keyboard, two integers separated by a whitespace. Then the statement

```
cin >> n >> m;
```

causes the computer to pause while characters are entered from the keyboard and stored in memory at locations identified by $n$ and $m$. It is equivalent to the following two statements.

```
cin >> n;          // first input from screen is stored in n
cin >> m;          // second input from screen is stored in m
```

Notice that a variable must be declared before it can be used. The integer variables $n$ and $m$ are declared by the statement:

```
int n, m;          // declare n and m to be integers
```

The character '$\backslash n$' represents a newline. A character string like "*Sum of integers from*" must appear between double quotation marks, while a single character appear between single quotation marks such as '$A$', '5' and '$\backslash n$'. Note that '$\backslash n$' is a single character. See §1.3.2 for more details on characters, §4.5 on strings, and §4.6 on input and output streams.

The words *int, double, if,* and *for* are reserved words in C++ and can not be used for the name of a variable. The reserved word *int* stands for integers while *double* stands for double precision floating point numbers. The variable *sum* is declared to have double precision by

```
double sum = 0.0;  // sum is initialized to 0
```

This statement not only declares *sum* to have type *double*, but also assigns it an initial value 0. A variable may not have to be initialized when it is declared.

In the *if* conditional statement

```
if (n > m) {       // if n is bigger than m, swap them
```

```
  int temp = n;     // declare temp and initialize it
  n = m;            // assign value of m to n
  m = temp;         // assign value of temp to m
}
```

the statements inside the braces will be executed if the condition $n > m$ ($n$ is strictly larger than $m$) is true and will be skipped otherwise. This *if* statement instructs the computer to swap (interchange) the values of $n$ and $m$ if $n$ is larger than $m$. Thus, after this *if* statement, $n$ stores the smaller of the two input integers and $m$ stores the larger. Notice that a temporary variable *temp* is used to swap $n$ and $m$. The braces in an *if* statement can be omitted when there is only one statement inside them. For example, the following two statements are equivalent.

```
if (n > m) m = n + 100;     // one statement
if (n > m) {                // an equivalent statement
  m = n + 100;
}
```

The second statement above can also be written in a more compact way:

```
if (n > m) { m = n + 100; }
```

The *for* loop in the sample program

```
for (int i = n; i <= m; i++) {
  sum += i;
}
```

first declares variable $i$ of type *int* and assigns the value of $n$ to $i$. If the value of $i$ is less than or equal to the value of $m$ (i.e., $i <= m$), the statement inside the braces $sum += i$ will be executed; Then statement $i{+}{+}$ (equivalent to $i = i + 1$ here) is executed and causes the value of $i$ to be incremented by 1. The statement $sum += i$ (equivalent to $sum = sum + i$, meaning $sum$ is incremented by the value of $i$) is executed until the condition $i <= m$ becomes false. Thus this *for* loop adds all integers from $n$ to $m$ and stores the result in the variable $sum$. The braces in a *for* loop can also be omitted if there is only one statement inside them. For example, the *for* loop above can also be written in a more compact way:

```
for (int i = n; i <= m; i++) sum += i;
```

Except for possible efficiency differences, this *for* loop can be equivalently written as

```
for (int i = n; i <= m; i = i + 1) sum = sum + i;
```

The compound operators in $i{+}{+}$ and $sum\ +=\ i$ can be more easily optimized by a compiler and more efficient than $i = i + 1$ and $sum = sum + i$. For example, an intermediate value of $i + 1$ is usually obtained and stored and then assigned to $i$ in $i = i+1$; while in $i{+}{+}$ such an intermediate process could be omitted and the value of $i$ is just incremented by 1. Notice that += is treated as one operator and there is no whitespace between + and =. See §2.3.7 for details on the *for* statement and §2.3.3 for compound operators.

Finally the value of *sum* is output to the screen by the statement:

```
cout << "Sum of integers from " << n << " to " << m
     << " is: " << sum << '\n';   // output sum to screen
```

First the character string "*Sum of integers from* " is output, then the value of $n$, and then string " *to* ", value of $m$, string " *is:* ", value of *sum*, and finally a newline character '\n'.

Suppose the above program is stored in a file called *sample.cc*. To compile it, at the UNIX or Linux command line, type

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

Then the machine-executable object code is written in a file called *add*. If you just type

```
c++ sample.cc
```

the object code will be automatically written in a file called *a.out* in UNIX (or Linux) and *sample.exe* in DOS. On other operating systems, compiling or running a program may just be a matter of clicking some buttons. Now you may type the name of the object code, namely, *add* or *a.out* at the UNIX command line, and input 1000 and 1. You shall see on the screen:

```
Enter two integers:
1000  1
Sum of integers from 1 to 1000 is: 500500
```

In this run, the input number 1000 is first stored in $n$ and 1 in $m$. Since $n$ is larger than $m$, the *if* statement interchanges the values of $n$ and $m$ so that $n = 1$ and $m = 1000$. The *for* loop adds all integers from 1 to 1000. Here is another run of the program:

```
Enter two integers:
1 1000
Sum of integers from 1 to 1000 is: 500500
```

In the second run, the input number 1 is stored in $n$ and 1000 in $m$. Since $n$ is smaller than $m$, the *if* statement is skipped.

A user may not have to input values from the terminal screen. Alternatively, this sample program can also be written as

```
#include <iostream>    // include input/output library
using namespace std;
```

```
main() {
  int n = 1;              // declare integer n with value 1
  int m = 1000;           // declare integer m with value 1000

  double sum = 0;
  for (int i = n; i <= m; i++) sum += i;
  cout << "The sum is: " << sum << '\n';
}
```

or in a more compact form:

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

main() {
  double sum = 0;
  for (int i = 1; i <= 1000; i++) sum += i;
  cout << "The sum is: " << sum << '\n';
}
```

The disadvantage of the alternative forms is that, if one wishes to add integers from 2 to 5050, for example, then the program needs to be modified and recompiled.

By convention, C++ files are usually postfixed with *.cc*, *.c*, *.C*, *.cpp*, and the like, and C++ compilers are *c++*, *g++*, *gcc*, *CC*, and so on.

Notice that unlike in C or FORTRAN, declarations (like *double sum*;) may not have to appear at the beginning of a program and can be mixed with other statements. The rule of thumb is that a variable should not be declared until it is used. This should increase the readability of a program and avoid possible misuse of variables.

## 1.2 Types and Declarations

Consider the mathematical formula:

```
z = y + f(x);
```

For this to make sense in C++, the *identifiers* $x$, $f$, $y$, and $z$ must be suitably declared. All identifiers must be declared before they can be used. *Declarations* introduce entities represented by identifiers and their types (e.g. *int* or *double*) into a C++ program. The types determine what kind of operations can be performed on them and how much storage they occupy in computer memory. For example, the declarations

```
int x;                 // x is declared to be of integer type
float y = 3.14;        // y is a floating point number
double z;              // z is a floating point number
```

```
double f(int);           // f is a function taking an integer as
                         // its argument and returning a double
```

will make the above formula meaningful, where $+$ is interpreted as adding two numbers, $=$ assigns the value on its right-hand side to the variable on its left-hand side, and $f(x)$ is interpreted as a function call returning the function value corresponding to argument $x$. The first declaration above introduces variable $x$ to be of type *int*. That is, $x$ can only store integers. The second declaration introduces variable $y$ to be of type $float$ (single precision floating point number) and assigns an initial value 3.14 to it. A simple function definition is:

```
double f(int i) {
  return (i*i + (i-1)*(i-1) + (i-2)*(i-2) - 5)/3.14;
}
```

The function $f()$ takes integer $i$ as input and returns a double precision number as output. It calculates the value of the mathematical expression $(i^2 + (i-1)^2 + (i-2)^2 - 5)/3.14$ for a given integer $i$. In C++, the symbols $+, -, *,$ and $/$ mean addition, subtraction, multiplication, and division, respectively. All these statements can be organized into a complete C++ program:

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

double f(int i) {            // function definition
  return (i*i + (i-1)*(i-1) + (i-2)*(i-2) - 5)/3.14;
}

main() {
  int x = 4;
  float y = 3.14;
  double z = y + f(x);
  cout << "The value of z is: " << z << '\n';
}
```

Note that the definition of the function $f()$ can not be put inside the function $main()$. See §3.8 for more details on functions.

Basic types are discussed in the next section. Additional types including functions and user-defined types are discussed in subsequent chapters.

## 1.3 Basic Types

### *1.3.1 Integer Types*

The integer types are *short int*, *int*, and *long int*. An *int* is of natural size for integer arithmetic on a given machine. Typically it is stored in one machine word, which is 4 bytes (32 bits) on most workstations and mainframes, and 2 bytes (16 bits) on many personal computers. A *long int* normally occupies more bytes (thus stores integers in a wider range) and *short int* fewer bytes (for integers in a smaller range).

Integers are stored in binary bit by bit in computer memory with the leading bit representing the sign of the integer: 0 for nonnegative integers and 1 for negative integers. For example, the nonnegative *int* 15 may be represented bit by bit on a machine with 4 bytes for *int* as

sign bit
↓

| | |
|---|---|
| 0 | 0000000 00000000 00000000 00001111 |

Note that whitespaces were inserted to easily see that it occupies 4 bytes, and the leading (leftmost) bit 0 signifies a nonnegative integer. Negative integers are normally represented in a slightly different way, called two's complement; see §2.2.4. Thus a computer with 32 bits for *int* can only store integers

$$-2^{31}, -2^{31}+1, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{31}-2, 2^{31}-1,$$

while a computer with 16 bits for *int* can only store integers

$$-2^{15}, -2^{15}+1, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{15}-2, 2^{15}-1.$$

Half of them are negative and the other half are nonnegative. Note that $2^{31} = 2147483648$ and $2^{15} = 32768$. Numbers out of the given range of integers on a machine will cause *integer overflow*. When an integer overflows, the computation typically continues but produces incorrect results (see Exercises 1.6.9 and 2.5.14 for two examples). Thus a programmer should make sure integer values are within the proper range. Use *long int* if necessary or other techniques (see Exercise 3.14.21 where digits of a large integer are stored in an array of integers) for very large integers. For example, a *long int* may occupy 6 bytes and be able to store integers:

$$-2^{47}, -2^{47}+1, \ldots, -2, -1, 0, 1, 2, \ldots, 2^{47}-2, 2^{47}-1.$$

Although the number of 8-bit bytes of storage for *short int*, *int*, and *long int* is machine-dependent, it is always given by the C++ operator *sizeof*. In general

$$sizeof(short\ int) \le sizeof(int) \le sizeof(long\ int).$$

On one machine, $sizeof(int) = 4$ and $sizeof(long\ int) = 6$, while on another, $sizeof(int) = sizeof(long\ int) = 4$. However, it is guaranteed that a *short int* has at least 16 bits and a *long int* has at least 32 bits.

A new type called *unsigned int* for nonnegative integers does not store the sign bit and thus stores larger integers than the plain *int*, which is also called *signed int*. If on a machine *signed int* holds integers from $-32768$ to 32767, then *unsigned int*, occupying the same number of bits but without storing the sign bit, will hold nonnegative integers from 0 to 65535. An integral type (*short int*, *int*, and *long int*) can be *signed* or *unsigned*. The keyword *long* is a synonym for *long int*, *short* for *short int*, *signed* for *signed int*, and *unsigned* for *unsigned int*. An *int* is always *signed*; that is, *int* and *signed int* always mean the same thing.

To know their number of bytes on your machine, compile and run the program:

```
#include <iostream>
using namespace std;
main() {
  cout << "number of bytes in short = "
       << sizeof(short) << '\n';
  cout << "number of bytes in long int = "
       << sizeof(long) << '\n';
  cout << "number of bytes in int = "
       << sizeof(int) << '\n';
  cout << "number of bytes in unsigned int = "
       << sizeof(unsigned int) << '\n';
}
```

From now on, statements such as

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

may not be explicitly included in example programs to save space and concentrate on more important features.

Standard conversions are performed when different types appear in arithmetic operations. Truncations occur when there is not enough space for converting one type into another. For example,

```
int i = 321;         // stored in sizeof(int) bytes
short ii = 321;      // stored in sizeof(short) bytes
long iii = i;        // implicit conversion from int to long
iii = long(i);       // explicit conversion from int to long
iii = i + ii;        // implicit conversion,
                     // it is same as iii =long(i) +long(ii)

iii = 123456789;     // a big integer
```

```
ii = short(iii);     // conversion from long to short
cout << ii;          // on one machine, output of ii is -13035
```

Note that explicit type conversion requires the use of the name of the type. For example, *long*($i$) converts explicitly an *int* $i$ to *long*. Explicit type conversion is also called *cast*. Notice that when the value of long integer $iii = 123456789$ is assigned to short integer $ii$, the space occupied by $ii$ (*sizeof*(*short*) bytes) may not be enough to hold the value of $iii$. Truncation may occur and lead to errors. For example, on one machine with $sizeof(short) = 2$, this results in $ii = -13035$. The negative sign is caused by the leading (leftmost) bit in the binary representation of $ii$ (after overflow), which happens to be 1.

The suffix $U$ is used to explicitly write unsigned integer constants and $L$ for long integer constants. For example, 5 is a *signed int*, $5U$ is an *unsigned int* and $5L$ is a *long int*. They may occupy different numbers of bytes in memory.

```
unsigned int i1 = 5U;     // 5U means unsigned integer 5
long int i2 = 5L;         // 5L means long integer 5
int i = 5;                // 5 means signed integer 5
```

By default, a number is a decimal number (base 10). A number preceded by 0 (zero) is octal (base 8) and a number preceded by $0x$ is hexadecimal (base 16). The letters $a, b, c, d, e$, and $f$, or their upper case equivalents, represent hexadecimal numbers $10, 11, 12, 13, 14$, and 15, respectively. For example, 63 is a decimal number, 077 is an octal number, and $0x3f$ is a hexadecimal number. Octal and hexadecimal numbers can be conveniently used to express bit patterns; see §6.4 for an example. See §4.6 on how to print them out.

### *1.3.2 Character Types*

A *char* variable is an integral type and is of natural size to represent a character (in English and other similar languages) appearing on a computer keyboard (it occupies one byte almost universally and is assumed always to be one byte in this book). A *char* can only store an integer that fits in one byte (thus one of $2^8 = 256$ values); it corresponds to a character in a character set, including (American) ASCII. For example, the integral value 98 corresponds to the character constant '*b*' in the ASCII character set. Character constants should appear between single quotation marks like '*A*', '5', '*d*', '*n*' (newline character), '*t*' (horizontal tab), '*v*' (vertical tab), '\0' (null character), '\\\\' (backslash character), '\\"' (double quotation character), and '\\''(single quotation character). Notice the special characters above that use a backslash \\ as an escape character. See Exercise 1.6.5 for a few more. For example,

```
char cc = 'A';            // assign character 'A' to cc.
```

```
                          //   In ASCII, cc = 65.
cc = '\n';                // assign a new value to cc.
                          //   In ASCII, cc = 10
int i = cc;               // i = 10, implicit type conversion
short ii = short(cc);     // ii = 10, explicit type conversion
```

A *char*, occupying 8 bits, can range from 0 to 255 or from $-128$ to 127, depending on the implementation. Fortunately, C++ guarantees that a *signed char* ranges at least from $-128$ up to 127 and an *unsigned char* at least from 0 up to 255. The types *char*, *signed char*, and *unsigned char* are three distinct types and the use of *char* could cause portability problems due to its implementation dependency. For example,

```
char c = 255;             // c has all 8 bits 1
int i = c;                // or: int i = int(c). Now i = ?
```

What is the value of $i$ now? The answer is undefined. On an SGI Challenge machine, a *char* is unsigned so that $i = 255$. On a SUN SPARC or an IBM PC, a *char* is signed so that $i = -1$. A signed integer with all bits equal to 1 (including the sign bit) represents the integer $-1$ in two's complement; see §2.2.4 for more details. Thus a *char* should be used primarily for characters, instead of small integers.

A *char* is output as a character rather than as a numeric value. For example, the program segment

```
char c = 'A';
int i = 'A';                                   // i = 65 in ASCII
cout << c << 'B' << i << "CD" << '\n' ;
```

outputs AB65CD to the screen (assuming the ASCII character set is used).

Inherited from the C programming language, a constant string always ends with a null character. For example, the string "CD" consists of three characters: C, D, and the null character (\0), and "$\backslash n$" (notice the double quotation marks) is a string of two characters: the newline character ($\backslash n$) and the null character (\0). In contrast, '$\backslash n$' (notice the single quotation marks) is a single character. This can be checked by the *sizeof* operator:

```
int i = sizeof("CD");                // i = 3
int j = sizeof("\n");                // j = 2
int k = sizeof('\n');                // k = 1
```

### 1.3.3 Floating Point Types

The floating point types are *float*, *double*, and *long double*, which correspond to single precision, double precision, and an extended double precision, respectively. The number of 8-bit bytes of storage for each of them is given by the operator *sizeof*. In general

$$sizeof(float) \leq sizeof(double) \leq sizeof(long\ double).$$

On many machines, they occupy 4, 8, and 12 bytes, respectively, and a *float* stores about 6 decimal places of precision, a *double* stores about 15 decimal places, and a *long double* about 18. A *long double* with 16 bytes can have 33 decimal places of precision. The possible values of a floating point type are described by *precision* and *range*. The *precision* describes the number of significant decimal digits that a floating point number carries, while the *range* describes the largest and smallest positive floating values that can be taken by a variable of that type. For example, on machines with 4 bytes for *floats*, 8 bytes for *doubles*, and 12 bytes for *long doubles*, the range of *float* is roughly from $10^{-38}$ to $10^{38}$, the range of *double* is roughly from $10^{-308}$ to $10^{308}$, and the range of *long double* from $10^{-4932}$ to $10^{4932}$. This means that, on such machines, a *float* number $f$ is roughly represented in the form

$$f = \pm 0.d_1 d_2 \cdots d_6 \times 10^n,$$

where $-38 \leq n \leq 38$ and $d_1 \neq 0$ (here $0.d_1 d_2 \cdots d_6$ is called the fractional part of $f$); a *double* number $d$ is roughly represented in the form

$$d = \pm 0.d_1 d_2 \cdots d_{15} \times 10^n,$$

where $-308 \leq n \leq 308$ and $d_1 \neq 0$; and a *long double* number $g$ in the form

$$g = \pm 0.d_1 d_2 \cdots d_{18} \times 10^n,$$

where $-4932 \leq n \leq 4932$ and $d_1 \neq 0$. On such machines, an *overflow* happens when a *float* number $f = \pm q \times 10^m$ $(0.1 \leq q < 1)$ with $m > 38$ and an *underflow* happens for $m < -38$. Similar definitions of overflow and underflow can be made for double and extended double precisions. The value of a variable is often set to zero when *underflow* occurs. The preceding discussion is true only roughly and can be made more precise in binary representation, which is normally used in the computer; see §1.4.

The numbers of bytes given as examples above are used to store floating point types. The IEEE (Institute for Electric and Electronic Engineers) standards currently require that at least 80 bits be used for doing internal computation (128 bits are used on some machines for *long double* calculations). When the results of a computation are stored or output, accuracy is often lost due to roundoff errors. For example, the statements

```
float fpi = atan(1)*4;                  // include <math.h>
double dpi = atan(1)*4;
long double ldpi = atan(1)*4;
```

give different precisions of the value $\pi$. Note the function call $atan(1)$ (for computing the arctangent of 1) gives the value $(\pi/4)$ of the mathematical function $arctan(x)$ with $x = 1$. Trigonometric and other functions are in the library *<math.h>*; see §3.11. To see their difference when being output to the screen, we can use the *precision* function to specify the accuracy and use the *width* function to specify the number of digits in the output (see §4.6 for more details). For example, the statements

```
cout.precision(30);        // include <iostream>
cout.width(30);            // output occupies 30 characters
cout << fpi << '\n';
cout.width(30);
cout  <<  dpi << '\n';
cout.width(30);
cout  <<  ldpi << '\n';
```

output the following to the screen (on my computer).

```
       3.1415927410125732421875
3.14159265358979311599796346854
3.14159265358979323851280895941
```

Compared to the exact value of $\pi$ : 3.14159265358979323846264338..., the *float* value *fpi* has 7-digit accuracy, the *double* value *dpi* has 16-digit, while the extended double value *ldpi* has 19-digit accuracy. They are all calculated using the same formula $atan(1) * 4$. The internal calculation is done in the same way and accuracy is lost when the result is stored in $fpi, dpi$, and *ldpi*.

The IEEE standard introduces two forms of infinity (*Inf* or *Infinity* in computer output): $+\infty$ and $-\infty$, for very large and small floating point numbers (when it makes sense to do so). For example, $x/0$ and $y+y$ give $\infty$ for a positive floating point number $x$ within range and the biggest floating point number $y$. So do $x+\infty$, $x*\infty$, and $\infty/x$. Here $\infty$ is understood to be $+\infty$. Similar explanations hold for $-\infty$. The standard also introduces the entity $NaN$ (*N*ot-*a*-*N*umber) to make debugging easier. Indeterminate operations such as $0.0/0.0$ and $\infty - \infty$ result in $NaN$. So does $x + NaN$. Run the program in Exercise 1.6.12 and you will see a situation where *Infinity* and *NaN* are among the output, which can be useful debugging information.

Variables of different types can be mixed in arithmetic operations and implicit and explicit conversions can be performed. For example,

```
double d = 3.14;
int n = 2;
int m = d + n;         // m = 5, implicit type conversion
int k = int(d) + n;    // k = 5, explicit type conversion
double c = d + n;      // c = 5.14, implicit type conversion
```

Note that in the addition $m = d + n$, integer $n$ is first implicitly promoted to floating point number 2.0 and the result $d + n = 5.14$ is then implicitly truncated into integer 5 for $m$, causing loss of accuracy.

By default, a floating point constant is of double precision. For example, the number 0.134, $1.34E{-}1$, $0.0134E1$, or $0.0134e1$ is taken to be a *double*, which occupies *sizeof*(*double*) bytes. Its *float* representation is $0.134F$, or $0.134f$, suffixed with $F$ or $f$, which occupies *sizeof*(*float*) bytes. The number after the letter $E$ or $e$ means the exponent of 10. For example, $1.34e{-}12$ means $1.34 \times 10^{-12}$ and $1.34e12$ means $1.34 \times 10^{12}$. This is the so-called *scientific notation* of real numbers.

The reason for providing several integer types, unsigned types, and floating point types is that there are significant differences in memory requirements, memory access times, and computation speeds for different types and that a programmer can take advantage of different hardware characteristics. The type *int* is most often used for integers and *double* for floating point numbers. Despite the fact that *float* usually requires less memory storage than *double*, arithmetics in *float* may not always be significantly or noticeably faster than in *double* for a particular C or C++ compiler, especially on modern computers; see §4.4.1 for an example and some explanations.

### 1.3.4 The Boolean Type

A Boolean, represented by the keyword *bool*, can have one of the two values: *true* and *false*, and is used to express the results of logical expressions (§2.2.3). For example,

```
bool flag = true;           // declare flag to be of bool

// ... some other code that might change the value of flag
double d = 3.14;
if (flag == false) d = 2.718;
```

The operator $==$ tests for equality of two quantities. The last statement above means that, if *flag* is equal to *false*, assign 2.718 to variable *d*.

By definition, *true* has the value 1 and *false* has the value 0 when converted to an integer. Conversely, nonzero integers can be implicitly converted to *true* and 0 to *false*. A *bool* variable occupies at least as much space as a *char*. For example,

```
bool b = 7;      // bool(7) is converted to true, so b = true
int i = true;    // int(true) is converted to 1, so i = 1
int m = b + i;   // m = 1 + 1 = 2
```

### 1.3.5 The Void Type

A type that has no type at all is denoted by *void*. It is syntactically a fundamental type, but can be used only as part of a more complicated type. It is used either to specify that a function does not return a value or as the base type for pointers to objects of unknown type. These points are explained later. See, for example, §3.8 and Exercise 3.14.24.

## 1.4 Numeric Limits

Machine-dependent aspects of a C++ implementation can be found in the standard library <*limits*>. For example, the function *numeric_limits* $<double>::max()$ gives the largest *double* and function *numeric_limits* $<int>::min()$ gives the smallest *int* that can be represented on a given computer. The library $<limits>$ defines *numeric_limits* $<T>$ as a template class (see Chapter 7) that has a type parameter $T$, where $T$ can be *float*, *double*, *long double*, *int*, *short int*, *long int*, *char*, and *unsigned* integers. When $T$ is, for example, *float*, then *numeric_limits* $<float>$ gives implementation-dependent numbers for *float*. The following program prints out information on *float*.

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

main () {
  cout << "largest float = "
       << numeric_limits<float>::max() << '\n';
  cout << "smallest float = "
       << numeric_limits<float>::min() << '\n';
  cout << "min exponent in binary = "
       << numeric_limits<float>::min_exponent << '\n';
  cout << "min exponent in decimal = "
       << numeric_limits<float>::min_exponent10 << '\n';
  cout << "max exponent in binary = "
       << numeric_limits<float>::max_exponent << '\n';
  cout << "max exponent in decimal = "
       << numeric_limits<float>::max_exponent10 << '\n';
  cout << "# of binary digits in mantissa: "
       << numeric_limits<float>::digits << '\n';
  cout << "# of decimal digits in mantissa: "
       << numeric_limits<float>::digits10 << '\n';
  cout << "base of exponent in float: "
       << numeric_limits<float>::radix << '\n';
  cout << "infinity in float: "
       << numeric_limits<float>::infinity() << '\n';
  cout << "float epsilon = "
       << numeric_limits<float>::epsilon() << '\n';
  cout << "float rounding error = "
```

```
       << numeric_limits<float>::round_error() << '\n';
  cout << "float rounding style = "
       << numeric_limits<float>::round_style << '\n';
}
```

Similarly, implementation-dependent information on *double* and *long double* may be obtained as

```
main() {
  double smallestDouble = numeric_limits<double>::min();
  double doubleEps = numeric_limits<double>::epsilon();
  long double largestLongDouble =
     numeric_limits<long double>::max();
  long double longDoubleEpsilon =
     numeric_limits<long double>::epsilon();
}
```

and information on *char* may be obtained as

```
main() {
  cout << "number of digits in char: "
       << numeric_limits<char>::digits << '\n';
  cout << "char is signed or not: "
       << numeric_limits<char>::is_signed << '\n';
  cout << "smallest char: "
       << numeric_limits<char>::min() << '\n';
  cout << "biggest char: "
       << numeric_limits<char>::max() << '\n';
  cout << "is char an integral type: "
       << numeric_limits<char>::is_integer << '\n';
}
```

Some explanations are given now to the terms *epsilon* and *mantissa*. A machine *epsilon* is the smallest positive floating point number such that $1 + epsilon \neq 1$. That is, any positive number smaller than it will be treated as zero when added to 1 in the computer. It is also called the *unit roundoff error*. When a floating point number $x$ is represented in binary as $x = \pm q \times 2^m$, where $q = 0.q_1q_2 \cdots q_n$ with $q_1 = 1$ and $q_i = 0$ or 1 for $i = 2, 3, \ldots, n$, and $m$ is an integer, then $q$ is called the *mantissa* of $x$, $m$ is called the *exponent* of $x$, and $n$ is the number of bits in the mantissa. Due to finite precision in computer representation, not all numbers, even within the range of the computer, can be represented exactly. A number that can be represented exactly on a computer is called a *machine number*. When a computer can not represent a number exactly, rounding, chopping, overflow, or underflow may occur.

On a hypothetical computer called $Marc$-32 (see [CK99, KC96]; a typical computer should be very similar to it if not exactly the same), one machine word (32 bits) is used to represent a single precision floating point number:

$$x = \pm 1.b_1 b_2 \cdots b_{23} \times 2^m.$$

The leftmost bit in the machine word is the sign bit (0 for nonnegative and 1 for negative), the next 8 bits are for the exponent $m$, and the rightmost 23 bits for the fractional part ($b_1$, $b_2$, ..., and $b_{23}$) of the mantissa. Notice that, to save one bit of memory, the leading bit 1 in the fractional part is not stored. It is usually called the *hidden bit*. The exponent $m$ takes values in the closed interval $[-127, 128]$. However, $m = -127$ is reserved for $\pm 0$, and $m = 128$ for $\pm\infty$ (if $b_1 = b_2 = \cdots = b_{23} = 1$) and $NaN$ (otherwise). Thus the exponent of a nonzero machine number must be in the range $-126 \leq m \leq 127$. On such a machine, the single precision machine *epsilon* is then $2^{-23} \approx 1.2\times 10^{-7}$, the smallest positive machine number is $2^{-126} \approx 1.2 \times 10^{-38}$, and the largest machine number is $(2 - 2^{-23}) \times 2^{127} \approx 3.4 \times 10^{38}$ (corresponding to $b_i = 1$ for $i = 1, 2, \ldots, 23$, and $m = 127$). In double precision, there are 52 bits allocated for the fractional part of the mantissa and 11 bits for the exponent, which results in the machine *epsilon* to be $2^{-52} \approx 2.22 \times 10^{-16}$, the smallest positive machine number $2^{-1022} \approx 2.25 \times 10^{-308}$, and the largest machine number $(2 - 2^{-52}) \times 2^{1023} \approx 1.798 \times 10^{308}$.

It can be shown (see [CK99, KC96]) that for any nonzero real number $x$ and its floating point machine representation $\bar{x}$ (assuming $x$ is within the range of the computer and thus no overflow or underflow occurs), there holds

$$\left| \frac{x - \bar{x}}{x} \right| \leq epsilon,$$

in the case of chopping, and

$$\left| \frac{x - \bar{x}}{x} \right| \leq \frac{1}{2} epsilon,$$

in the case of rounding to the nearest machine number. That is, the relative roundoff error in representing a real number in the range of a computer with a particular precision is no larger than *epsilon* in that precision.

The machine *epsilon* and number of bits in binary (or equivalent digits in decimal) for representing the mantissa of a floating point number on a particular computer are given in the template class *numeric_limits*< *T*> for a floating point type $T$.

In addition, C library <*limits.h*> has macros (see §3.1 for some rules and examples for defining macros) such as those listed in Table 1.1, and C library <*float.h*> has macros such as those listed in Table 1.2. These macros are better avoided. However, the C++ library <*limits*> may not be available on early (nonstandard) compilers. In this case, numeric limits can be obtained as in the following program.

| | |
|---|---|
| $INT\_MAX$ | largest $int$ |
| $INT\_MIN$ | smallest $int$ |
| $LONG\_MAX$ | largest $long\ int$ |
| $LONG\_MIN$ | smallest $long\ int$ |
| $ULONG\_MAX$ | largest $unsigned\ long\ int$ |
| $UINT\_MAX$ | largest $unsigned\ int$ |
| $SHRT\_MAX$ | largest $short\ int$ |
| $USHRT\_MAX$ | largest $unsigned\ short\ int$ |
| $SCHAR\_MIN$ | smallest $signed\ char$ |
| $UCHAR\_MAX$ | largest $unsigned\ char$ |
| $CHAR\_MAX$ | largest $char$ |
| $CHAR\_MIN$ | smallest $char$ |
| $WORD\_BIT$ | number of bits in one word |
| $CHAR\_BIT$ | number of bits in $char$ |

TABLE 1.1. Limits of integral numbers from C library <limits.h>.

| | |
|---|---|
| $DBL\_MAX$ | largest $double$ |
| $DBL\_MIN$ | smallest $double$ |
| $DBL\_EPSILON$ | $double$ epsilon |
| $DBL\_MANT\_DIG$ | number of binary bits in mantissa |
| $DBL\_DIG$ | number of decimal digits in mantissa |
| $DBL\_MAX\_10\_EXP$ | largest exponent |
| $LDBL\_MAX$ | largest $long\ double$ |
| $LDBL\_MIN$ | smallest $long\ double$ |
| $LDBL\_EPSILON$ | $long\ double$ epsilon |
| $LDBL\_MANT\_DIG$ | number of binary bits in mantissa |
| $LDBL\_DIG$ | number of decimal digits in mantissa |
| $LDBL\_MAX\_10\_EXP$ | largest exponent |
| $FLT\_MAX$ | largest $float$ |
| $FLT\_MIN$ | smallest $float$ |
| $FLT\_EPSILON$ | $float$ epsilon |
| $FLT\_MANT\_DIG$ | number of binary bits in mantissa |
| $FLT\_DIG$ | number of decimal digits in mantissa |
| $FLT\_MAX\_10\_EXP$ | largest exponent |

TABLE 1.2. Limits of floating point numbers from C library <float.h>.

```
#include <limits.h>
#include <float.h>

main() {
  int i = INT_MIN;                        // smallest int
  long j = LONG_MAX;                      // largest long int
  double x = DBL_MAX;                     // biggest double
  long double y = LDBL_MAX;               // biggest long double
  float z = FLT_MAX;                      // biggest float
  double epsdbl = DBL_EPSILON;            // double epsilon
  float epsflt = FLT_EPSILON;             // float epsilon
  long double epsldb = LDBL_EPSILON;  // long double epsilon
}
```

For headers ending with the *.h* suffix such as *float.h* and *math.h*, the include directive

```
#include <float.h>
```

automatically gives access to declarations in *float.h*, and the statement

```
using namespace std;
```

is not necessary.

Note that the C++ library *<limits>* is different from the C library *<limits.h>*, which C++ inherited from C. All C libraries can be called from a C++ program provided that their headers are appropriately included. See §4.2 for more details on all C and C++ standard header files.

## 1.5 Identifiers and Keywords

### *1.5.1 Identifiers*

In our first program, we have used variables such as *sum* and *m* to hold values of different data types. The name of a variable must be a valid identifier. An identifier in C++ is a sequence of letters, digits, and the underscore character _. A letter or underscore must be the first character of an identifier. Upper and lower case letters are treated as being distinct. Some identifiers are:

```
double sum = 0;     // identifier sum suggests a summation
double product;     // identifier product suggests multiply
bool flag;          // flag certain condition (true or false)

int Count;
int count;                          // different from Count
int this_unusually_long_identifier; // legal, but too long
```

| | | | |
|---|---|---|---|
| **and** | **and_eq** | **asm** | **auto** |
| **bitand** | **bitor** | **bool** | **break** |
| **case** | **catch** | **char** | **class** |
| **compl** | **const** | **const_cast** | **continue** |
| **default** | **delete** | **do** | **double** |
| **dynamic_cast** | **else** | **enum** | **explicit** |
| **export** | **extern** | **false** | **float** |
| **for** | **friend** | **goto** | **if** |
| **inline** | **int** | **long** | **mutable** |
| **namespace** | **new** | **not** | **not_eq** |
| **operator** | **or** | **or_eq** | **private** |
| **protected** | **public** | **register** | **reinterpret_cast** |
| **return** | **short** | **signed** | **sizeof** |
| **static** | **static_cast** | **struct** | **switch** |
| **template** | **this** | **throw** | **true** |
| **try** | **typedef** | **typeid** | **typename** |
| **union** | **unsigned** | **using** | **virtual** |
| **void** | **volatile** | **wchar_t** | **while** |
| **xor** | **xor_eq** | | |

TABLE 1.3. All C++ keywords.

```
int _aAbBcCdD;                    // legal, but no meaning
```

It is good programming practice to choose identifiers that have mnemonic significance so that they contribute to the readability and documentation of a program. Confusing identifiers should be avoided. For example, *l1, ll, lo*, and *lO* are different and valid identifiers but are hard to read, and identifiers *Count* and *count* can be easily misunderstood.

### *1.5.2 Keywords*

Keywords such as *int*, *double*, and *for* are explicitly reserved identifiers that have a strict meaning in C++. They can not be redefined or used in other contexts. For example, a programmer can not declare a variable with the name *double*. A keyword is also called a reserved word. A complete list of C++ keywords is in Table 1.3.

Note that *cin* and *cout*, for example, are not keywords. They are part of the input and output (I/O) library *<iostream>*.

## 1.6 Exercises

1.6.1. Modify the program in §1.1 to compute the sum of the squares of all integers between two given integers. That is, find the sum $n^2 + (n+1)^2 + \cdots + m^2$ for two given integers $n$ and $m$ with $n < m$. In the sample program in §1.1, the variable *sum* should be declared as *double* or *long double* in order to handle large values of $n$ and $m$. In this exercise, try to compute *sum* in two ways: as a *long double* and as an *int*, and compare the results. On a computer with 4 bytes for storing *int*, the second way calculates the sum $1^2+2^2+3^2+\cdots+5000^2$ as $-1270505460$. Why could a negative number as the output be possible?

1.6.2. Modify the program in §1.1 to multiply all integers between two given small positive (e.g., less than or equal to 12) integers. When one of them is 1 and the other is a positive integer $n$, the program should find the factorial $n! = 1 * 2 * 3 * \cdots * (n-1) * n$. The program to find $n!$ may be written as

```
#include <iostream>
using namespace std;
main() {
  int n;
  cout << "Enter a positive integer: \n";
  cin >> n;

  int fac = 1;
  for (int i = 2; i <= n; i++) fac *= i;   // fac = fac*i;
  cout << n << "! is: " << fac << '\n';
}
```

Except for possible efficiency difference, the statement $fac *= i$ is equivalent to $fac = fac * i$.

This simple program can only compute $n!$ correctly for $n = 1$ up to $n = 12$ on a computer with $sizeof(int) = 4$. When $n = 13$, such a computer may calculate 13! as 1932053504 while the correct value is $13! = 6227020800$. It may also compute 20! as $-2102132736$ (why negative?) and 40! as 0 (why just 0?). A computer can produce garbage very easily. Test what your computer gives you for $13!, 20!$, and 40!. A user should check the correctness of computer outputs by all means. Sometimes outputs such as the erroneous result for 13! above can be very hard to check. Outline a procedure to determine the correctness of your computer output of 13!, assuming you do not know the correct value.

In Exercise 3.14.21, a technique is introduced that can compute $n!$ correctly for much larger $n$, for example, $n = 3000$ or larger. Notice that the number 3000! has 9131 digits that would overflow as an integer on any current computer.

1.6.3. If one wishes to compute the following summation

$$\sin(1.1) + \sin(1.3) + \sin(1.5) + \cdots + \sin(9.9),$$

the program in §1.1 can be modified to do so:

```
#include <iostream>            // input/output library
#include <math.h>              // math library for sin

main() {
  double sum = 0;              // sum initialized to 0

  for (double d = 1.1; d <= 9.9; d += 0.2) sum += sin(d);
  cout << "The sum is: " << sum << '\n';
}
```

This *for* loop declares $d$ to be a variable of double precision with initial value 1.1, and executes the statement $sum \mathrel{+}= sin(d)$ for $d$ changing from 1.1 up to 9.9 with increment 0.2 each time. Compile and run this program, and modify it to compute

$$e^{1.1} + e^{1.2} + e^{1.3} + \cdots + e^{15.5},$$

where $e$ is the base of the natural logarithm. See §3.11 for a complete list of mathematical functions in the library <*math.h*>.

1.6.4. Write a program that outputs the largest integer, smallest integer, and the number of bytes for storing an integer on your computer. Also do this with *short int*, *int*, and *long int*.

1.6.5. Write a program that outputs the integer values corresponding to characters $A$, 9, $a$, {, \$, \\$n$ (new line), \\$t$ (horizontal tab), \0 (null character), \\\\ (backslash), \\$r$ (carriage return), \" (double quote), \\$b$ (backspace), \\$f$ (formfeed), \' (single quote), \\$v$ (vertical tab), \? (question mark), and \\$a$ (alert) on your computer.

1.6.6. Write a program that outputs exactly the sentences:
He said: "I can output double quotation marks."
She asked: "Do you know how to output the newline character \\$n$ ?"

Notice that the double quotation mark, newline, and question mark are special characters in C++. See Exercise 1.6.5

1.6.7. A backslash symbol \ at the end of a line is supposed to continue it to the next line. Test the following program to see its effect.

```
#include <iostream>
using namespace std;
main() {
  cout << "I am outputting a string that stands on \
three lines to test the effect of a continuation line \
using a backslash\n";
}
```

1.6.8. Write a program that outputs the largest and smallest numbers, *epsilon*, the number of decimal digits used to store the mantissa, and the largest exponent for double precision floating point numbers on your computer. Repeat this for *float* and *long double* as well.

1.6.9. Compile and run the program

```
#include <iostream>
using namespace std;
main() {
  long g = 12345678912345;
  short h = g;           // beware of integer overflow
  int i = g - h;
  cout << "long int g = " << g << '\n';
  cout << "short int h = " << h << '\n';
  cout << "their difference g - h = " << g - h << '\n';
}
```

on your computer. Does your compiler warn you about *integer overflow*? It may not always do so. Does your computer give $g - h = 0$? Beware of overflow and truncation in converting a larger type to a smaller one.

1.6.10. Calculate the value of $\pi$ on your computer following the steps in §1.3.3 in single, double, and extended double precisions. How many digits of accuracy does your computer give in each of the three floating point types?

1.6.11. What is the value of $i$ in the following statement?

```
int i = 3.8 + 3.8;
```

Is it 6, 7, or 8? Type conversion may lead to loss of accuracy. Does your compiler warn you about this? It may not always do so.

1.6.12. What do you think the following program will output?

```
#include <iostream>
#include <float.h>

int main() {
  double x = DBL_MAX;                   // biggest double
  double epsilon = DBL_EPSILON;         // double epsilon
  double zero = 0.0;
  double y = 100.2;

  double z0 = x + x;
  double z1 = x * 2;
  double z2 = epsilon/9;
  double z3 = y/zero;
  double z4 = zero/zero;
  double z5 = z3 - z3;
  double z6 = x + y;

  cout << "outputting results:\n";
  cout << z0 << '\n';
  cout << z1 << '\n';
  cout << z2 << '\n';
  cout << z3 << '\n';
  cout << z4 << '\n';
  cout << z5 << '\n';
  cout << z6 << '\n';
  cout << 1 + z2 << '\n';

}
```

Run the program on your computer to check the results. You may see Infinity, NaN, and other unusual outputs.