1 Building programs in a Linux environment

1.1 The editor, the compiler, and the make system
   1.1.1 Troubleshooting mysterious problems
1.2 A quick tour of input and output
1.3 Where to find information on the C++ standard library
1.4 Command line arguments and return values
1.5 Obtaining numerical constants from input strings
1.6 Resolving shared libraries at run time
1.7 Compiling programs from multiple units
1.8 Libraries and library tools
1.9 More on Makefile
1.10 The subversion source code management system (SVN)
   1.10.1 The SVN repository
   1.10.2 Importing a project into SVN
   1.10.3 The basic idea
1.11 Style guide: advice for beginners
1.12 Exercises
Bibliography

The goal of this book is to develop computational skills and apply them to problems in physics and the physical sciences. This gives us a certain license to try to teach any (legal) computational skill that we believe will be useful to you sooner or later in your career. The skill set you’ll need includes scientific computing and not-specifically-scientific computing. For example, applying statistical techniques in data analysis or solving the Schrödinger equation on the computer are distinctly scientific computing tasks, whereas learning how to work collaboratively with a code management system is a not-specifically-scientific task. But you will use both as you progress in your career and so we will aim to teach you a little of both. We will go back and forth to some extent between scientific computing topics and general computing topics, so that the more generic skill set becomes useful in writing programs of a scientific nature, and the scientific programs provide opportunities to apply and reinforce the full skill set of numerical and not-so-numerical techniques.
Building programs in a Linux environment

Like mathematics, computing often appears to be a collection of tricks, with the well-known tricks elevated to the status of techniques. Deciding which tricks and techniques to teach is a difficult question, and a book like this has no traditional road-map. Our selection criterion is usefulness. Many of the topics are concerned with the simulation, classification and modeling of experimental data. Others (like integration or computational linear algebra) provide a basis for some of the later topics in simulation and modeling. Later, application to classical and quantum mechanical problems will be discussed. Like mathematics, computation is an art, and as practitioners we will pass on our own approach. If you learn to play a saxophone from John Coltrane, you will be absorbing John Coltrane’s style, but also, hopefully, developing your own style along the way. So it is with the art of computation.

Writing executable programs and toolkits is of course central to this enterprise; since we are not about to describe computational techniques in wholly abstract terms, we have to be specific about which language(s) we are proposing to use. Our coding examples are usually expressed in the modern C++ language, or occasionally in the older, simpler computing language “C”. We will sometimes also employ “pseudocode”, which is a generic code-like description of any algorithmic process. Like almost any choice in computation, the focus of C++ is not totally obvious or universally acclaimed, but rather involves certain pros and cons—a debate that we will not lead you through here. The motivation for our choice is:

- In physics our lives consist of manipulating objects which are more abstract than scalar data types, including vectors, spinors, matrices, group elements, etc. While calculations involving these objects can be done in many computer languages, our lives will be vastly simpler if our computer languages support the objects of day-to-day life. No language is vast enough to support these at the language level, but languages supporting the object-oriented paradigm do allow you to add user-defined objects to the set of built-in data types. C++ also allows us to define basic operations on these data types, and maintains the speed of a compiled language.

- Most of a typical operating system is written in C and you will find it very easy to integrate specifically scientific software together with a vast body of generic software, particularly lower-level system calls.

Few people learn how to write software by writing programs from the bottom up. The “software stack” of even a very simple program can already involve toolkits that have taken a generation or two of computer scientists and physicists to develop. It is very common to make big scientific contributions by working on a small part of a huge program. Making modifications to an existing program, or filling in a piece of an incomplete program, can be a valuable learning experience. Some infrastructure for building programs is generally required. At a very minimum, a computing platform, operating system, and suite of compilers is needed. More complicated projects may even require a sophisticated set of tools to coordinate the development, distribution, and build of a software system. As more and more software is packaged and distributed for re-use,
the build of computer programs becomes more challenging. In this chapter we introduce
the basic ideas related to the building of software with some very simple examples.

Our reference operating systems are the popular Ubuntu Linux (now at version 17.04)
and macOS (version 10.12.6). These are both variants of Unix, an operating system
written in C and dating back to the 1970s; and we will refer to them generically as
such. The commands required for writing, building, and executing programs as well
as tailoring the environment will be expressed as required by the bash shell on a Ubuntu
Linux machine. Because of its low cost and portability, the Linux operating system is
widely used in scientific computing. Not only can it be used on personal computers
(laptops, desktops, and now even smart phones), but it can also be found running in
the machine rooms of large computer centers on thousands of high-density rack mount
computers. The latest version of Ubuntu Linux can always be installed on a PC after
downloading from the website www.ubuntu.com. The installation, management and
customization of operating systems are not trivial skills, but they are also extremely useful.

Your first task is to get a laptop or a PC, and equip it with a basic Linux operating
system. We recommend that you install and maintain the operating system yourself.
It is possible to dual-boot desktop and/or laptop computers, preserving the original
operating system (e.g. Windows) which then coexists with Linux. A Macintosh
computer, which runs MacOS, will also do for this book, since it runs an operating
system similar to Linux.

We assume a working knowledge of C++ basics—the part of C++ which is essentially
just the C programming language, but minus anachronisms such as malloc, free,
printf, scanf. In this text we develop in a few chapters that which is necessary to
go beyond the ground level and understand classes, inheritance, polymorphism, and
templates. For those who need to brush up on the basics, a good, short, but somewhat
anachronistic introduction is the famous text of Kernighan and Ritchie (1988). The first
few chapters of Capper (1994) or Bronson (2013) also cover the basics and provide
a more modern introduction to the same subject. Another good source is the tutorial
section of the online reference www.cplusplus.com. While our presentation of the
C++ language will be far less formal than other common treatments of this powerful
computing language, the physics applications will be more interesting and appropriate
for the physical sciences, and you will “learn by doing”, though it may be a good idea to
refer to the above references occasionally if you prefer a more formal treatment of the
language.

1.1 The editor, the compiler, and the make system

You write a program with an editor. There are a large number of these available on
Linux, and in principle any one will do. The Emacs text editor (provided by the GNU
Table 1.1  List of programs (center column) commonly used to compile major computer languages (left column).

<table>
<thead>
<tr>
<th>Language</th>
<th>Compiler under linux, OS X</th>
<th>Provided by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fortran</td>
<td>gfortran</td>
<td>GNU Project</td>
</tr>
<tr>
<td>C</td>
<td>cc gcc</td>
<td>GNU Project</td>
</tr>
<tr>
<td>C++</td>
<td>c++ g++</td>
<td>GNU Project</td>
</tr>
<tr>
<td>Java</td>
<td>javac</td>
<td>Oracle</td>
</tr>
</tbody>
</table>

project; homepage www.gnu.org/s/emacs/) has features such as syntax highlighting, which can be very helpful in writing code, since it can recognize and draw to your attention syntax errors so that you can recognize them before they are reported by the compiler. The gedit text editor (provided by the GNOME project, homepage projects.gnome.org/gedit/) also has some of these features, and is perhaps more intuitive though less powerful. A much more basic editor called vi is generally pre-installed on even the most basic linux distributions. This is the editor of choice for gnarled veterans. On the other end of the spectrum, interactive development environments such as Eclipse (provided by the Eclipse foundation, homepage www.eclipse.org) embed powerful editors in a suite of tools for code development in C++ as well as other languages. There can be a steep learning curve with Interactive Development Environments (IDEs) such as Eclipse, but the effort can be worthwhile.

A single file of instructions is called a compilation unit. A compiler turns these into object code, and a linker puts different pieces of object code together into an executable program. Each computing language (Fortran, C, C++, Java) has its own compiler. Since C++ is a superset of C we can and will use the C++ compiler everywhere. Under linux, g++ and c++ are the same program. A table of common compilers is given in Table 1.1.

We look at a simple program which is a single compilation unit. It has a routine called main and like other functions takes arguments and returns an integer value (more on that, later). We call the file containing these lines foo.cpp. “.cpp” is the most common extension for c++ code. The program illustrates the important features of the main program unit, particularly how to write new commands under a unix operating system.

```
int main (int argc, char ** argv) {
    return 0;
}
```

Here are three ways to build an executable program from this source:

1. Compile to object code and then link to make an executable program.

```
$c++ -c foo.cpp -o foo.o
$c++ foo.o -o foo
```
2. Compile/link at the same time to make an executable program in one step.

```bash
$c++ foo.cpp -o foo
```

3. Use make

```bash
$make foo
```

The compilation step transforms human-readable C++ into machine instructions that the CPU can understand, and is called object code. The link step links together object code from various sources into an executable program. Which sources? The example above may give you the impression that there is only one, called `foo.cpp` but that is not true. Your program also contains pieces from the **C standard library** `libc.so` as well as others.

Even when the compilation and link is performed in one single command, there are still two phases to the process, and thus two points of failure. If you get an error message, try to figure out whether the error message is a compile error or a link error. Link errors do not occur at specific instructions, but constitute a failure to assemble the final program from the various pieces of object code, and usually in this case a piece of the program, or the object code containing the piece has been omitted, is missing, or cannot be located.

Once you’ve built the program you can see which run time libraries have been linked by issuing the command:

```bash
$ldd foo
```

which will generate the output

```
linux-vdso.so.1 => (0x00007fffc77fe000)
libc.so.6 => /lib/x86_64-linux-gnu/libc.so.6 (0x00007f19f6313000)
/lib64/ld-linux-x86-64.so.2 (0x00007)
```

In addition to these libraries that are included automatically in the link of any C++ program, additional libraries can be linked by mentioning them explicitly in the list of arguments to the compiler as we will soon see, using the `-l` and `-L` flags. In general programs will include program libraries containing useful functions and class libraries containing useful classes (extensions to the basic data types of the language).

Our first example (`foo.cpp`) is extremely simple and not at all a typical project, which these days can consist of many thousands of compilation units. Managing the development and build of large software infrastructure becomes a complicated job. Usually the `make` system (provided by the GNU project, homepage `http://www.gnu.org/software/make/`) is used to build projects containing multiple compilation units. A pedagogical guide can be found in Mecklenburg (2004). The third way of building the program `foo` illustrates the basic principle of the system: `make` knows that a program (`foo`) can be built from its sources (`foo.cpp`) with `g++`, the C++ compiler.
The editor, the compiler, and the make system

It applies a set of rules to build the target from its prerequisites. One of those rules says that the g++ command can be used to build an executable with name foo from a source code file named foo.cpp.

The make system can be extensively customized (and usually is) whenever make is used in the context of a large software project. The customization is achieved through Makefiles, which are files usually named makefile, Makefile, or GNUMakefile that are placed in directories that contain the source code. In some cases these makefiles are written by hand and in others they may be generated automatically by another tool. We will describe this further as the need arises. Powerful as make is, it is often not sufficient on its own to organize the build of large or complicated projects, so additionally, a number of code management systems are available to coordinate and manage the distributed development of software on top of make.

1.1.1 Troubleshooting mysterious problems

On a few occasions you are likely to find that properly written code does not compile, link, or execute because of the configuration of the platform and not the code itself. Header files (with .h or extensions, discussed below) are normally installed in a system directory such as /usr/include or /usr/local/include; they can also be installed in other locations but then the qualifier

-I/path/to/include/area

must be added to the command line during the compile step. If the header files are not installed there then obviously the compile step will fail. Libraries, discussed in Section 1.8, are specified during the link step with the -l flag, and their search path is specified using the -L flag. These libraries must exist, they must be located during the link step, they must actually contain the symbols that they are supposed to provide, and they must be compatible. These symbols are all of the local and global variables known to the compiled code, as well as all the known structures, classes, free subroutines and member functions.

Computing hardware and operating systems exist in both 32 bit and 64 bit architectures, and object code which has been compiled for a 64 bit machine will generally not run on a 32 bit machine. Normally this type of object code would not be built or installed on the wrong architecture, but computers are machines and machines can go wrong. You might find a program or object library on a cross-mounted disk drive shared by machines having different architectures. Even the execution step can fail if bits of object code collected in shared libraries (files with the .so extension, discussed below) do not exist, cannot be located, or are incompatible. Incompatibilities can sometimes be caused by linking together object code produced by different compilers, or even the same compiler with different options. If these problems arise, the best approach is to be systematic in investigating and determining the cause.

To that end, it’s useful to know about a few utilities in unix to help debug mysterious problems with “perfectly good” code. Table 1.2 summarizes a few of them to help you
### Table 1.2 Table of utilities for examining executable files. This is useful for investigating “mysterious problems” as described in the text.

<table>
<thead>
<tr>
<th>Linux command</th>
<th>OS X equivalent</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldd</td>
<td>otool -L</td>
<td>check shared libraries required by a program. Prints the location of these libraries and flags any missing libraries.</td>
</tr>
<tr>
<td>file</td>
<td>file</td>
<td>classifies files. This command can yield useful information about how object files, and libraries, were compiled.</td>
</tr>
<tr>
<td>nm</td>
<td>nm</td>
<td>lists symbols in the object files, archives, shared libraries, and executables (variables and subroutines). Output can be very long!</td>
</tr>
<tr>
<td>c++filt</td>
<td>c++filt</td>
<td>Decodes “mangled” C++ symbols (for example, from nm or link error messages) and prints them in human-readable form.</td>
</tr>
</tbody>
</table>

see what you have just built; the unix manual pages for these commands give more information.

### 1.2 A quick tour of input and output

The first program you will generally write, like the famous Hello, World example in Kernighan and Ritchie (1988), simply echoes a few words to the terminal. In Fortran, input/output (IO) is handled at the language level; in C it is handled at the level of standard C functions, and in C++ it is handled through objects. Three important objects you must learn about are `std::cout` (the standard output), `std::cerr` (the standard error output), and `std::cin` (the standard input). In fact, since C++ is a superset of C, the C standard library routines (`printf` and `scanf` for the cognoscenti) can also be used but they are considered obsolete and should be avoided because they are unable to handle user-defined data types.

Basic usage of the `std::cin`, `std::cout`, and `std::cerr` objects is extremely easy. You will need to include the `iostream` header file at the top of any compilation unit that uses them:

```cpp
#include <iostream>
```

Consider the following line:

```cpp
std::cout << "Hello, World" << std::endl;
```
The “<<” characters in the above line constitute an operator, the left shift operator, which is defined in the C++ standard library for the \texttt{std::cout} object. The operator can stream bits of text, \texttt{ints}, \texttt{floats}, and \texttt{doubles}, and even user-defined data types (if the designer allows it) to the standard output, i.e. the terminal. You should try this on your own. This is all we will say about \texttt{std::cout} for the moment. About \texttt{std::cerr}, we will say only that under unix operating systems (i.e. linux, Mac OS, and other variants), it is convenient sometimes to have two streams, both of which will normally end up printing to the terminal, because it is possible to redirect each stream separately\footnote{for more information, see the unix manual page on bash, particularly the section REDIRECTION.} to a file or a unix pipe, for example. So, \texttt{std::cerr} functions just like \texttt{std::cout} except that normally program output is sent to \texttt{std::cout} while informational status, warning, and error messages are sent to \texttt{std::cerr}.

Your program can read input from the terminal using the \texttt{std::cin} class. This class can read in bits of text but also read the values of \texttt{int}, \texttt{float}, \texttt{double} (among others) from text strings from standard input–normally you think of typing these in using your actual fingers, but under unix you can also tell a program to take its “standard input” from a file, like this:

```bash
$ foo < file.txt
```

We use \texttt{std::cin} as follows in our programs:

```cpp
#include <iostream>
...
int i, float f;
std::cin >> i >> f;
```

Input has one unique subtlety which we need to discuss: it can fail for several reasons. One reason is that the input may not be convertible to the right data type. For example, the word “particle” cannot be interpreted as an integer. Another reason is that the input may have reached its end, if the input were a file, or if the typing were terminated by C\textasciitilde D (Control+D). So we normally test that input has worked with the following incantation:

```cpp
int i;
if (std::cin >> i) { // success!
    ...
} else { // failure!
    ...
}
```
1.3 Where to find information on the C++ standard library

The objects \texttt{std::cout}, \texttt{std::cerr}, and \texttt{std::cin} are all part of the C++ standard library, which contains a very large set of functions and objects, many of which will be extremely useful. The most usable documentation that we have encountered so far is an online reference, \texttt{www.cplusplus.com}, particularly the section called “library reference”.

The C++ standard library is beyond the scope of our treatment and we will not attempt to explain more than the essentials, just what you’ll need to follow this text and complete the exercises. We use it throughout the text, and discuss certain aspects in more detail in Chapters 6, 12, and 17. When you need detailed information, the ultimate authority on the C++ language and the C++ run time library is the ISO C++ standard. An updated standard was published in August 2011 by the International Standards Organization (ISO) and the International Electrotechnical Commission (IEC) as report number ISO/IEC 14882:2011. While this document is authoritative, it has little pedagogical value and is really intended for experts.

The best advice we can give to students confronted with such a large and complex set of standard software tools is to read the documentation as you go along, and try to learn, along the way, some of the tricks that will make you more efficient in your work. In this text we will explain bits and pieces of the C++ standard library as we go along, without any intent or commitment to treat the subject comprehensively.

1.4 Command line arguments and return values

An executable program is, generally speaking, a command like any other command in unix. Like any command, you can pass in \textbf{command-line arguments}. These are available through the variable \texttt{char **argv}, an array of character strings, whose length is given by \texttt{int argc}. Let’s modify our program \texttt{foo} now so that it simply echoes the command line arguments:

```
#include <iostream> // Include headers for basic i/o

int main (int argc, char ** argv) { //
    for (int i=0;i<argc; i++) { // Loop over command line args
        std::cout << argv[i] << " "; // Print each argument to screen.
    }
    std::cout << std::endl; // End-of-line.
    return 0; // Program successful
}
```
If you build this program and execute it with a few command-line arguments, it will behave as follows:

$ ./foo A B C D
./foo A B C D

Notice that the zeroth argument is the command name itself. The $ symbol in the preceding example is the command prompt, echoed by the shell unless the user has configured his system otherwise.

If you are new to unix, you may not know that the operating system looks for commands in a standard set of directories. An ordered list of directories is held in the environment variable PATH. To see what directories are in your path, you can type

$ echo $PATH
/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin:
/usr/games

For that matter, since you have now written a program very much like echo, you can use it to discover your path as well:

$ ./foo $PATH
./foo /usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

You can create your own directories where you can create important programs that you have written:

$ mkdir ~/bin
$ cp ./foo ~/bin
$ export PATH=~/bin:$PATH

This latter command can be added to the .bashrc file in your home directory, so that it is executed every time you log in. Then the program foo will appear as a built-in command, just like any other unix command, which, in fact, are built for the most part in exactly the same way. The source code for every command in Linux is free and publicly available—and most are written in the C programming language. For example, the echo command, which echoes the command line in a way very similar to foo, is compiled from source code which can be seen in http://git.savannah.gnu.org/cgit/coreutils.git/tree/src/echo.c; this web page points to a software repository which also contains other linux commands that may be familiar to you. You now have a partial glimpse of how a unix operating system, particularly many of the commands, are constructed.

Now back to our program, foo. The main routine of our program returns an integer value, zero in our example. This is the return status of the program. It’s value can be
accessed after the program has executed by the “special” parameter “?”. Its value is decoded by prepending a “$”, as for any environment variable (try it! type echo $?). It is typical to signal successful completion of a command by returning 0, and one or more failure modes with a nonzero value.

1.5 Obtaining numerical constants from input strings

One way of inputting numerical constants to a program is to read them from standard input. This is convenient in some circumstances but not in others, since from the point of view of the user of the program (who is also often the developer) the most useful way to communicate input to the program may be via the command line. However, as we have seen, command line arguments are made available as an array of character strings. How do we extract numbers from these strings?

Part of the C++ standard library is a class called std::istringstream. To use this object, you initialize it with a character string and then extract the numerical data through the right shift operator, >>. As with the object std::cin, which is closely related to std::istringstream objects, you can test for success. Here is an example of how to parse an integer from the second item on the command line:

```c++
#include <sstream>
...
int main (int argc, char **argv) {
    ...
    int anInteger;
    std::istringstream stream(argv[1])
    if (stream >> anInteger) { // Success!
        ...
    }
    else { // Failure!
        ...
    }
}
```

1.6 Resolving shared libraries at run time

Note that most programs are not complete without a set of shared libraries that are required by the program. Those shared libraries are to be found, normally, in standard locations. By default the area /usr/lib is searched for these libraries first, then /lib. By defining an environment variable called LD_LIBRARY_PATH, which is an ordered list of colon-separated directories, you can specify other libraries to search. If a shared
library cannot be found, the system will report an error when you attempt to execute a
program:

$myprogram
myprogram: error while loading shared libraries:
libMyLibrary.so.1: cannot open shared object file: No such file
or directory

Missing libraries will also be reported by the ldd utility, described in section 1.1.

1.7 Compiling programs from multiple units

In textbooks and exercises, programs consist of a few lines of code typed into a single
source code file. In the real world of physical sciences, computations this simple are rare. In
the context of large scientific experiments, often the software infrastructure
is a behemoth consisting of multiple millions of lines of code split across tens of
thousands of individual files, representing a major financial investment on the part of
national governments. This is the ecosystem in which students in the physical sciences,
particularly physics and astrophysics, may find themselves trying to operate. It should
be quite obvious that organizing and managing software infrastructure involves breaking
it down into more manageably sized units. In this and the following sections we will see
how to compile a program from multiple compilation units, and then how to archive the
units and coordinate the build of the program.

While the necessity of splitting up source code arises from very complicated systems,
we will illustrate the basic idea by splitting up a small one. Our program, called iterate,
is designed to make a certain number of calls to the system routine sleep. It takes two
command line parameters: the number of iterations, and the duration of each iteration:

$ iterate 6 4
0 sleeping for 4 seconds
1 sleeping for 4 seconds
2 sleeping for 4 seconds
3 sleeping for 4 seconds
4 sleeping for 4 seconds
5 sleeping for 4 seconds
$

Here is the program, all in one single file, which we call iterate.cpp

```cpp
#include <iostream>    // for std::cout, & cetera
#include <sstream>     // for istringstream
#include <cstdlib>     // for exit
```

//
/ Define a data structure
/
struct Control {
    int iterations;
    int seconds;
};
/
// Parse the command line:
/
Control *initialize (int argc, char ** argv) {
    Control *control=NULL;
    if (argc!=3) {
        std::cerr << "Usage: " << argv[0] << " iterations seconds"
            << std::endl; exit(0);
    } else {
        control =new Control;
        control->iterations=0;
        control->seconds=0;
        {
            std::istringstream stream(argv[1]);
            if (!(stream >> control->iterations)) return control;
        }
        {
            std::istringstream stream(argv[2]);
            if (!(stream >> control->seconds)) return control;
        }
        return control;
    }
    // finalize:
    //
    void finalize(Control *control) {
        delete control;
    }
    //
    // execute:
    //
    void execute(Control *control) {
        if (control) {
            for (int i=0;i<control->iterations;i++) {
                sleep(control->seconds);
            }
        }
    }
The simplest way to build this is to type make iterate, which will in turn execute the following command in a child process:

$g++ iterate.cpp -o iterate

To illustrate the typical way of organizing a project with multiple compilation units, we will separate iterate.cpp into several pieces. The main program declares and defines three functions (in addition to main): initialize, execute, and finalize. It also declares a single data structure, called Control. The functions are defined where they are declared, which must be before they are used. Hence main is the last function to be defined, rather than the first.

In C++, as in C, every function and data structure (or class) must be declared before it is used, once, and only once; this amounts to specifying the interface to the function and/or class. The first step to breaking up this program is to separate the declarations from the definitions, putting them in a header file that can be included by each compilation unit. We therefore now create the header file iterate.h, and put it in the same directory as the source code:

```c
#include <iostream>

int main(int argc, char ** argv) {
  Control *control = initialize(argc, argv);
  execute(control);
  finalize(control);
  return 0;
}
```

```c
// Data structure controlling the iteration loop:
 struct Control {
   int iterations;
   int seconds;

   // Initialize. Parse the command line:
   Control *initialize(int argc, char ** argv);
```
// Execute the iteration loop:
void execute(Control *control);

// Finalize. Clean up the memory:
void finalize(Control *control);

#include "iterate.h"
#include <iostream>

void execute(Control *control) {
    if (control) {
        for (int i=0;i<control->iterations;i++) {
            sleep(control->seconds);
            std::cout << i
                     << " sleeping for "
                     << control->seconds
                     << " seconds"
                     << std::endl;
        }
    }
}

The functions initialize and finalize have been moved into their own files, in a similar way; and the program iterate.cpp now becomes very simple:

#include "iterate.h"

int main(int argc, char ** argv) {
    Control *control = initialize(argc, argv);
    execute(control);
    finalize(control);
    return 0;
}

At this stage our directory contains four source code files—one of them containing the function main, which is required in any program. We can build the program iterate
in a number of different ways from these sources. The first option that we have is to compile and link all of the source code at the same time:

```bash
$g++ iterate.cpp initialize.cpp execute.cpp finalize.cpp \
   -o iterate
```

This is fine except it has the drawback that if you change one part of this “project” you need to recompile all of its files. This is not too high a price for the program `iterate`, but could be very high in a real-world project. The preferred option is to compile each unit to object code separately, and then link them together at the end. The `g++` command can be used for both stages of this procedure (here, the `-c` flag indicates that `g++` should compile to object code and not bother to attempt a link):

```bash
$g++ -c -o iterate.o iterate.cpp
$g++ -c -o initialize.o initialize.cpp
$g++ -c -o execute.o execute.cpp
$g++ -c -o finalize.o finalize.cpp
$g++ iterate.o initialize.o execute.o finalize.o -o iterate
```

A mixed approach can also be taken:

```bash
$g++ -c -o iterate.o iterate.cpp
$g++ -c -o initialize.o initialize.cpp
$g++ iterate.o initialize.o execute.cpp finalize.cpp -o iterate
```

Stepping back and looking at what we have done, we notice that the modularity of the program is greatly improved; the protocol for each routine is clear from looking at the header file (where additional documentation can also be collected), and developers can work individually on one piece of the program at a time. On the other hand, while the program development itself is simplified, the price is additional complexity on the side of building the program. In the real world the management of very many compilation units is an extremely complicated task often carried out by teams of full-time people. Fortunately, a number of standard tools have been developed to better manage the build of a program. In the following section we will discuss several of them that are almost always at the core of any project: libraries of object code, the make system, and a source code management system.

## 1.8 Libraries and library tools

Having seen how to break up the source code into more manageable units, we now address the issue of how to keep the compiled code together in order to ease the process of compilation. Files containing object code (with the `.o` extension) are often collected together into libraries, so that the client code (`iterate.cpp` in the above
example) can link with one library rather than many different object files. In the world of professional programming the library (and associated header files and documentation) is the implementation of what is called the API, or *application programming interface*, which represents a toolkit for the development of applications, vulgarly referred to as “apps” amongst the rabble.

There are two kinds of libraries, called static (files ending with `.a`) and shared (files ending with `.so` on Linux systems or with `.dylib` on the Mac). The main difference is that at link time, the object code in static libraries becomes part of the executable file; i.e., it is copied in. Programs linked to a shared library contain only pointers to bits of executable code which are not copied into the executable; and therefore the shared library file must be resolved at run time, before the program can be executed (see Section 1.6). Working with static libraries is usually simpler though it does create larger executables.

The Linux command `ar` manipulates a static library. To create the static library `libIterate.a`, give the command:

```
$ar rc libIterate.a initialize.o execute.o finalize.o
```

The modifier `rc` stands for “replace” and “create”: the library (or archive) is created if it doesn’t exist, and if it does exist then any object files within the library are replaced by those given on the command line. The contents of the library can be examined with `ar t`:

```
$ar t libIterate.a
initialize.o
execute.o
finalize.o
```

Also, the contents of the library can be manipulated; for example individual pieces of object code may be extracted (`ar x`) or deleted (`ar d`). More information can be obtained from the `ar` manual page.

Now that you’ve built the library, you naturally will wonder how to link it to other code. The `g++` command has two important compiler flags that can be used, `-L` and `-l`. The first one, `-L`, specifies a directory to be searched for the library. The working directory is not on that path by default, so if your library lives in your working directory you should add `-L .` or `-L `pwd` ` to the `g++` command. The second one, `-l`, gives the name of the library to link, but with an odd rule: to link to `libIterate.a`, you should write `-lIterate`. In other words, transform “lib” into `-l` and drop the extension `.a`. Both flags can be repeated on the command line and specify a search list: if the library is not found in the first directory, the second is searched; likewise if the symbols are not found in the first library on the command line, the second is searched. Therefore the order of `-L` and `-l` arguments is important. In our example, you will use the command:

```
$g++ iterate.o -L . -lIterate -o iterate
```
To make a shared library, you can use the versatile g++ command again, giving the list of object files on the command line in addition to the -shared flag, and specifying a lib prefix and an .so extension for the output file, like this:

```
$g++ -shared initialize.o execute.o finalize.o -o libIterate.so
```

Some compilers also require the -fPIC qualifier during the compilation of object code files like initialize.o in order to produce “relocatable” code required in a shared library\(^2\). The only way to examine the contents of a shared library file is with the nm command, which dumps the symbol table. It is usually best to pipe this through c++filt, so that subroutine names are demangled into a readable form. You can link to the shared library in the same way as you link to a static library, and by default libIterate.so (for example) will be take precedence over libIterate.a if both files are found. Typically, when you then run the program, the shared object library will need to be resolved as well, as we have discussed in Section 1.6.

The software on a linux system is built from packages and these packages include programs, APIs, and sometimes both programs and APIs. One good (and useful) example is the gnu scientific library, or gsl, which can be installed (on a Ubuntu linux system) by typing:

```
$sudo apt-get install libgsl0-dev
```

This installs the libraries /usr/lib/libgsl.a, /usr/lib/libgsl.so, as well as the headers /usr/include/gsl/*.*h, and a manual page that can be referenced by typing

```
$man gsl
```

### 1.9 More on Makefile

The example we have been developing over the last few sections now constitutes a mini-project with a number of steps in the build process, namely (if we are going to use a static library):

- build the object files initialize.o, execute.o, finalize.o
- build the archive libIterate.a
- build the executable iterate

Extrapolating our experience to a large project, we can see that while building each constituent requires knowledge of a few compiler flags at the very least, building the entire

\(^2\) This is particularly the case with the gnu compiler g++.\)
Building programs in a Linux environment 19

project interactively quickly becomes prohibitive. One way to automate this is to script
the build (in bash, csh, or a similar scripting language); however with large projects
one prefers to skip compiling code that has not changed since the last compilation. The
make system was invented to build components of a large project from prerequisites.
It can detect when a component is stale (because its prerequisites have changed) and
rebuild it. It does this from rules, some of them built-in, while others can be specified
to the system. The typical way to extend the set of built-in rules is to write a Makefile
which lives in the same directory as the source code. We demonstrate this by an example.
First, notice that the make system already knows that a .cpp file is prerequisite to an .o
file with the same base name; typing make initialize.o is sufficient for the system
to execute

```
g++ -c -o initialize.o initialize.cpp
```

when initialize.cpp is found in the working directory and has been modified more
recently than initialize.o. Ditto for execute.o and finalize.o. Makefile does
not know (yet) that these object files are prerequisite to the archive libIterate.a.
Thus, we create a file named Makefile and add the following lines:

```
libIterate.a:initialize.o execute.o finalize.o
```

This says that libIterate.a is a target and that it depends on the three listed object
files. Since it is the first target in the Makefile (and the only one for the moment), typing
make will automatically build the three prerequisite files, from built-in rules. Following
this line in the Makefile, one can specify which actions to take to build the target. Such
lines must begin with a tab. The make system is very sensitive to this and using spaces
will cause make to fail. We revise the Makefile so that it looks like this:

```
libIterate.a:initialize.o execute.o finalize.o
	ar rc libIterate.a initialize.o execute.o finalize.o
```

This can also be written a little differently, since the syntax of Makefile allows you to use
the expression $@, which means “the target”; $? which means “the list of prerequisites”;
and “$<”, which means “the first prerequisite”. Thus one can write:

```
libIterate.a:initialize.o execute.o finalize.o
	ar rc $@ $?
```

Macros can be defined in a Makefile too, so one can write this as:

```
OFILES=initialize.o execute.o finalize.o
libIterate.a:$ (OFILES)
	ar rc $@ $?
```
Additional targets can be added to this list which now only includes \texttt{libIterate.a}. A common one to add is \texttt{clean} which removes all compiled code (.o files, static and/or shared libraries, the executable) and sometimes the backup files (ending in ~) left by the emacs editor. Such a line would look like

\begin{verbatim}
clean:
\texttt{rm -f *.a *.o iterate *~}
\end{verbatim}

(Obviously you must be extremely careful!) and we can clean our directory now by typing \texttt{make clean}, which will rid the directory of everything but source code. Now we still have not told the make system how to build the final executable. We insert (at the top, after macro definition but before any other target) a line that determines how the executable \texttt{iterate} will be built; because this line will be the first target in the \texttt{Makefile}, \texttt{iterate} will be the \texttt{default target}. It will be built merely by typing \texttt{make} from that directory. The final version of \texttt{Makefile} looks like this:

\begin{verbatim}
OFILES=initialize.o execute.o finalize.o

iterate:iterate.o libIterate.a
  g++ -o $@ $< -L. -lIterate

libIterate.a:$(OFILES)
  ar rc $@ $?

clean:
  rm -f *.a *.o iterate
\end{verbatim}

You can build the whole project by typing \texttt{make}, or pieces of it:

\texttt{
$make execute.o
$make libIterate.a
}

for example.

This probably looks like a nice tool, but even \texttt{Makefile} has its limits and many code management systems have been built on top of \texttt{Makefile} when things get even more complicated. We will not discuss these in this book, but be prepared for a shock when you enter the world of million-line software projects.

\subsection{1.10 The subversion source code management system (SVN)}

The last important tool that we will discuss in this introduction is the \textit{Subversion system}, or \textit{SVN}, which is very widely used to manage the process of distributed software development. This is an example of a source code management system. Other examples
include the Concurrent Version System (CVS), which is now practically obsolete and git which is more recent and rapidly gaining popularity. In some ways these systems resemble a dropbox service, which is perhaps more familiar to a general audience. While a source code management system provides a central synchronized repository for code, it contains special features that enable tracking source code development and even restoring an entire project to a specific version and/or date. Some of the exercises in this book will require you to reposit the assignment in SVN; and your instructor will provide access to an SVN repository on a remote machine. Besides the ability to track changes and restore the code to some previous state, SVN allows for multiple developers to work on the same piece of code. Alternatively, the same individual can work on a project from multiple locations. This, for example, allows you to move potentially valuable pieces of work (including, but not limited to code, perhaps your thesis!) off of your laptop computer and onto a central site. Like many of the tools discussed in this chapter, SVN has a lot of powerful functionality and we will describe just enough of it so that you can get started using it. We assume that for your first experience with the system, somebody else (i.e. your instructor) will set up the SVN repository for you, set up the necessary access mechanisms, and grant you the necessary access rights.

1.10.1 The SVN repository

A repository is an abstract concept that implies a central place where code resides. How central? Repositories can be set up and administered by unprivileged users as files on a single laptop computer, if desired; more privileged administrators can configure servers where contributions from users around the country or around the world are centralized; and now commercial “cloud computing” services operate servers that can be accessed by users at little or no expense.

In the first case the SVN repository is a single file that the user creates:

\$svnadmin create /path/to/repositoryfile

This creates a repository in a file called /path/to/repositoryfile that can now be filled, but never by acting directly on the repository file, which is henceforth only to be modified using SVN commands; i.e., those with the format svn command options. One useful command, svn ls, simply lists the contents of the repository, which, if you have just created it as explained above, is empty. The syntax of the command is

\$svn ls file:///path/to/repositoryfile

If you wish to execute this from a remote machine (assuming that both local and remote machines can communicate via the secure shell, ssh), you can issue the following command from the remote machine:

\$svn ls svn+ssh://user@host.domain/path/to/repositoryfile
where user@host.domain specified the username, hostname, and domain name of the local machine. Other types of servers, using other protocols, may have other prefixes such as http://, https://, or svn://.

For the following examples we assume that you have access to an SVN repository because either

- You have created a repository, as described above.
- You have requested and received a repository through a web-hosting service such as Sourceforge (www.sourceforge.com) or Cloudforge (www.cloudforge.com).
- Your instructor has set up a repository via one of the above methods, and given you access and instructions.

Web-hosting services have interesting benefits, first because they require no effort from the users to maintain and administer, second because they often add additional useful tools such as web-based code browsing. Whatever solution is adopted will refer to the repository in the following as protocol://repository.

1.10.2 Importing a project into SVN

A project can be imported to SVN as follows. First we assume that the original source code lives in a directory called /path/to/PROJECT. To import that code into SVN issue the following command:

$svn import /path/to/PROJECT protocol://repository/PROJECT

This creates a project within the repository, and copies all of the files in /path/to/PROJECT into the repository. The project then grows only when additional directories are added to it, additional files are added to directories, or files are modified.

Each change to the repository is logged, and all the logged information is accessible via the command svn log. If you execute the svn import command as it appears above, an editor window will pop up and prompt you for a log message. Alternately, short log messages can be added by adding the option -m "here is my log message" to the command line.

1.10.3 The basic idea

We now have both a repository and a project residing in that repository. With this initialization out of the way, we describe the basic SVN operations for source code management.

The main idea behind SVN is that the repository holds a master copy of the project, while developers check out a local copy to their machine and make modifications. When they are satisfied, they check their modifications back into the master copy (repository). This is called a copy-modify-merge model. The simplest scenario is illustrated in Figure 1.1. The command
Building programs in a Linux environment

Figure 1.1 Sketch of the simplest SVN transaction; first a checkout, then a commit following modification to the local copy.

$svn checkout protocol://repository/PROJECT.

checks out the package (step 1 in Figure 1.1), and the command

$svn commit -m "message"

puts the modifications back into the repository (and adds the message “message” to the log). This is all that is needed as long as the file content of the project does not change. If files need to be added to the project then this is done with $svn add file1 [file2] [file3] ...; the command schedules files for addition but does not add them until $svn commit is issued; directories are added in the same way. $svn remove file1 [file2] [file3] ...can be issued to remove files from the repository (after the files are removed from the local copy).

With multiple local copies in play (e.g., with multiple developers) the situation is more complicated because the master copy can change while one of the developers is making modifications to his or her local copy. Figure 1.2 shows a diagram of a typical transaction.

1. User 1 checks out the package to his/her local copy.
2. User 2 checks out the package to his/her local copy.
3. User 2 commits a modification to the repository. At this point user 1’s local copy is out of date. User 1 will not be able to commit until he refreshes his own copy with the changes that have gone into the repository.
4. User 1 refreshes by typing $svn update.
5. Then user 1 checks his changes into the repository with $svn commit, after resolving any conflicts.
Conflicts can arise when two users simultaneously work on the same package, but usually SVN is smart enough to resolve those without user intervention. SVN will resolve situations in which two users work on two different files in the same project. It will also resolve conflicts when two users have altered different parts of the same file. The only situation that normally requires user intervention is when two users have changed the same part of the same file; this can occur when two people jump in to fix the same bug. Conflicts are clearly indicated in the text and generally do not cause much trouble. We refer the reader to Collins-Sussman et al. (2004) for a detailed description of what to do when these cases arise.

The entire set of operations is carried out with very few SVN commands: checkout, commit, update, add, remove, status, and log; which suffices for most needs. The `svn checkout` and `update` commands have options that let users checkout, or update to, a tagged version, or a version that was current on a specific date and time. The `SVN log` command allows one to view the whole revision history of a file, including all of the comment messages that were given as arguments to the commit command. Finally, `svn status` displays the status of the project as a whole and/or individual file, depending on the parameters to the command.

It is important to never introduce executable programs, libraries, object code, backup copies, or large binary data files into the repository. These will make the administration of the repository difficult. Besides, these files need not be stored in the repository since they are typically regenerated from source. Putting `Makefiles` in the repository to aid in the build is generally a good idea.
After this brief introduction, we leave the reader to explore the full set of SVN operations using the references and/or the exercises.

1.11 Style guide: advice for beginners

We close with an offering of advice for beginners that has been gleaned over 80 combined years of programming experience. Presumably these good practices will be set by protocols if you are working in a large group. If you are not, read on.

dive in. Learning a computer language should be like learning French or German. You don’t need to have mastered all the nuances of the plus-que-parfait tense to order a burger in Paris. Look at examples, learn some basics, and get coding!

adopt a convention. Adopting a convention for naming your files, libraries, variables, objects, and classes is important. Established conventions do exist, but we will not venture to advocate one over another since this is a matter of personal choice (or a choice that has been imposed by management). Peruse the Wikipedia article on Hungarian notation to get an idea of the options, and controversy, in adopting conventions.

use descriptive names. Calling an index \( i \) is traditional, but it is much better to give it a descriptive name such as \( \text{particleNumber} \).

document your code. You will not remember the inner workings of your code after a few months. Describe its purpose, method, important variables, compiler directives, and usage somewhere (often at the top of the main file). If you are not using a repository, record revision history.

write clear code. You might think this is obvious, but it is quite easy for code to become unclear and one must be vigilant. Sooner or later you will find some clever compact way to do something and add it to your code. But you will not remember the trick the next time you look at your program and will not feel so clever. If you are going to be clever at least document it.

At a more general level, clear code is the result of clearly understanding the computational issue. Of course, as scientific programmers it is your duty to understand your problem to the best of your ability.

Lastly, if your code is destined to be used by others you should also consider the typical user’s conceptualization of the code. In particular, you want the user interface to match the user’s expectations. A familiar version of this problem is when a user attempts to change an aspect of text in a word processing program and all sorts of unexpected and frustrating code behavior ensues.

trap errors. A part of structured programming is trapping and dealing with errors when they are generated. This is implemented, for example, in C++ or java with the try and catch statements. Unfortunately, no such functionality exists in C or Fortran. Do what you can.
avoid magic numbers. Strings of digits should not appear in your code. You may recognize 0.197327 as $\hbar c$ in the appropriate units but the next person to work on your code may not. Better to place a statement such as `static float hbarc = 0.197327` somewhere. While we are on this, why would you ever write `double pi = 3.1415926` when you could write `double pi = 4.0*atan(1.0)`, or even better include the header `<cmath>`, which defines many mathematical constants, including $\mathbb{M}_\mathbb{P}_\mathbb{I}$?

allow for parameter input. Students tend to code parameter values directly into their programs. In our experience, programs are never run once and it is therefore important to allow for flexible parameter input. This is typically accomplished in one of three ways: (i) query the user for console input, (ii) via the command line, (iii) via an input file. Option (i) is simple; option (ii) permits easy scripting; and option (iii) is useful when many parameters are required but is more difficult to manage. One approach is to query the user and echo input data into a file. This file can then be modified and redirected as input in subsequent runs: `./foo < input.txt`. A more sophisticated approach might use a graphical user interface or a database.

specify default parameters. It is easy to spend days or weeks determining parameter values that optimize sought behavior. Once these are found they should be documented with the project, preferably as defaults in the code itself. We guarantee that you will not remember the preferred temporal grid spacing in your electromagnetic field computation a few months from now.

document the output. Most scientific code produces output files, sometimes dozens of them. These files should contain header information that specifies the program that created them, parameter values that were used, and the meaning (and units) of the output.

learn a debugger. Simple programs can be debugged with judicious print statements. However more complicated issues will require the use of a debugger, such as `gdb`. The GNU collaboration has created a free graphical front end for a variety of debuggers called `DataDisplayDebugger` that we happily recommend. Integrated Development Environments (such as Eclipse) often have debugging capability as well. It is worth the effort to learn to use these.

squash kludge. As you develop code you will often find yourself adding options to deal with an ever-increasing number of cases. The result is a kludgy mess with multiple layers of `if` or `case` statements. At this stage it is better to redesign the program or, more simply, to clone variant code.

develop systematically. Plan the general code structure before writing anything. Develop code in small logical chunks and debug as you go. Rushing ahead is a guarantee of future headache.
test extensively. If you are writing scientific code, you are doing science, and your code must be reliable. Check your results against known limits and special cases with analytical solutions. If you must strike out on your own, try to make the change from known territory as minimal as possible (for example, substituting a complicated potential for a square well potential in a quantum mechanics problem). Debugging Monte Carlo code is extremely difficult! You need analytic limits and simple cases worked out for comparison.

understand code parameters. Students often confuse physical parameters and algorithmic parameters. The former define the physics problem that you want to solve. The latter specify the method in which you solve the problem—your answer should be independent of algorithmic parameters. It is up to you to confirm this! In practice this means extrapolating to infinity or zero. You can always double or halve an algorithmic parameter and confirm stability of your answer. Even better is to plot how your answer changes with algorithmic parameters, and better yet is to have a theoretical understanding of the functional form of that dependence so that reliable extrapolations can be made.

Similar advice with additional detail is contained in Wilson (2014).

1.12 Exercises

1. Write and compile an empty program. Name it exp . cpp, and put in in a directory called CH1/EX1. Compile it, and use the four utilities in Table 1.2 to examine the output file. Add to your empty program a call to the math library function exp, which requires you to include the header file:

```
#include <cmath>
```

Run the nm and the ldd utilities on the program and describe carefully what happens before and after the call to exp is added. Try this with both a constant and a variable argument to exp. Pipe the output of nm through c++filt and explain how the output is different. (This is called “name mangling” and “demangling”).

2. Now write a variant of this program in a directory called CH1/EX2. Add to your program a single command line argument. Your program should now accept a single command line argument and echo the exponential of that argument, like this:

```
$exp 5
148.413
```

Examine the program with ldd and nm. Pipe the output of nm through c++filt.
Exercises

3. In a directory called CH1/EX3, replace the call to your own version of the exp function. Use a Taylor series expansion\(^3\). Check again with \texttt{ldd}, \texttt{nm} and \texttt{nm | c++filt}. Tabulate \(x\) vs. \(\exp(x)\) for \(x = \{-10, -8, -6, \ldots, 6, 8, 10\}\). By switching between your version of \(\exp\), and the one in the math library, determine how \texttt{nm} tells you whether the \(\exp\) routine is internal or external to the executable program file.

4. In unix the wildcard character is the star or asterisk, "\(*\)". In a directory called CH1/EX4, write the program \texttt{foo} described in Section 1.4. Go into your home directory and execute \texttt{/path/to/foo \*}. Explain the output.

5. Write a program called \texttt{plus} (in a directory called CH1/EX5) which expects two integers on the command line, adds the two integers and prints out the answer. The expected behavior of this program is:

\[
\$ \texttt{plus 3 7} \\
3 + 7 \Rightarrow 10
\]

6. In the previous example, break the code up into three compilation units:

   a) \texttt{main.cpp} – main program
   b) \texttt{parse.cpp} – parses the command line
   c) \texttt{add.cpp} – adds the two input values
   d) \texttt{print.cpp} – prints the result

Put these four files together a directory called CH1/EX6, and add a Makefile which builds:

   a) a library called \texttt{libPlus.a}
   b) an executable called \texttt{plus}

Make sure that the Makefile contains a “clean” target to remove all object code and executables generated by the compiler during the make procedure.

7. Clone the directory CH1/EX6 into a directory CH1/EX7; then modify your Makefile so that the shared library \texttt{libPlus.so} is used, rather than a static library. Check and report the size of the final executable in both circumstances.

8. Write Makefiles in the directories (CH1/EX1-CH1/EX7) to build all of the targets in the previous exercises. Make sure each executable has a “clean” target.

9. Your instructor will provide access to a SVN repository. Add the source code and Makefiles for CH1/EX1-CH1/EX7 to the repository.

   a) make sure that you do not reposit any object code.
   b) make sure that you can check out and build each executable from scratch.

\(^3\) In Chapter 4 we will learn a better way to implement an exponential function
BIBLIOGRAPHY