

CPL was a joint project between Cambridge University and Imperial College in London. Initially, the project had been done in Cambridge, so “C” officially stood for “Cambridge.” When Imperial College became a partner, the official explanation of the “C” became “Combined.” In reality (or so we are told), it always stood for “Christopher” after Christopher Strachey, CPL’s main designer.

## References

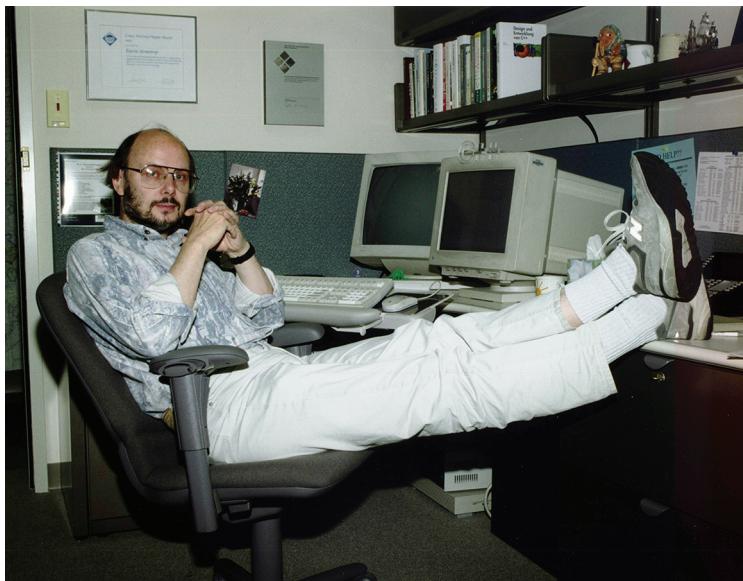
- Brian Kernighan’s home pages: <http://cm.bell-labs.com/cm/cs/who/bwk> and [www.cs.princeton.edu/~bwk/](http://www.cs.princeton.edu/~bwk/).
- Dennis Ritchie’s home page: <http://cm.bell-labs.com/cm/cs/who/dmr>.
- ISO/IEIC 9899:1999. *Programming Languages – C*. (The C standard.)
- Kernighan, Brian, and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1978. Second Edition, 1988. ISBN 0131103628.
- A list of members of the Bell Labs’ Computer Science Research Center: <http://cm.bell-labs.com/cm/cs/alumni.html>.
- Ritchards, Martin. *BCPL – The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
- Ritchie, Dennis. “The Development of the C Programming Language. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Salus, Peter. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.

### 22.2.6 C++

C++ is a general-purpose programming language with a bias toward systems programming that

- Is a better C
- Supports data abstraction
- Supports object-oriented programming
- Supports generic programming

It was originally designed and implemented by Bjarne Stroustrup in Bell Telephone Laboratories’ Computer Science Research Center in Murray Hill, New Jersey, that is, down the corridor from Dennis Ritchie, Brian Kernighan, Ken Thompson, Doug McIlroy, and other Unix greats.



Bjarne Stroustrup received a master's degree (in mathematics with computer science) from the university in his hometown, Aarhus in Denmark. Then he went to Cambridge, where he got his Ph.D. (in computer science) working for David Wheeler. The main contributions of C++ were to

- Make abstraction techniques affordable and manageable for mainstream projects
- Pioneer the use of object-oriented and generic programming techniques in application areas where efficiency is a premium

Before C++, these techniques (often sloppily lumped together under the label of “object-oriented programming”) were mostly unknown in the industry. As with scientific programming before Fortran and systems programming before C, it was “well known” that these techniques were too expensive for real-world use and also too complicated for “ordinary programmers” to master.

The work on C++ started in 1979 and led to a commercial release in 1985. After its initial design and implementation, Bjarne Stroustrup developed it further together with friends at Bell Labs and elsewhere until its standardization officially started in 1990. Since then, the definition of C++ has been developed by first ANSI (the national standards body for the United States) and since 1991 by ISO (the international standards organization). Bjarne Stroustrup has taken a major

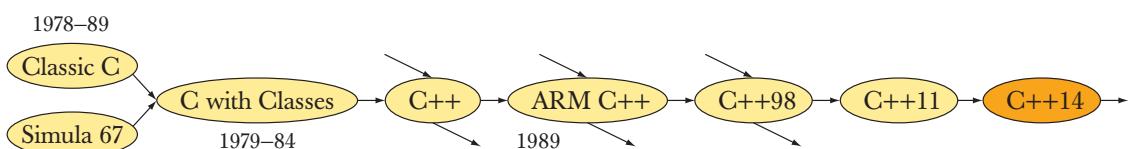
part in that effort as the chairman of the key subgroup in charge of new language features. The first international standard (C++98) was ratified in 1998 and the second in 2011 (C++11). The next ISO standard will be C++14, and the one after that, sometimes referred to as C++1y, may become C++17.

The most significant development in C++ after its initial decade of growth was the STL – the standard library’s facilities for containers and algorithms. It was the outcome of work – primarily by Alexander Stepanov – over decades aiming at producing the most general and efficient software, inspired by the beauty and utility of mathematics.



Alex Stepanov is the inventor of the STL and a pioneer of generic programming. He is a graduate of the University of Moscow and has worked on robotics, algorithms, and more, using a variety of languages (including Ada, Scheme, and C++). Since 1979, he has worked in U.S. academia and industry, notably at GE Labs, AT&T Bell Labs, Hewlett-Packard, Silicon Graphics, and Adobe.

The C++ family tree looks like this:



“C with Classes” was Bjarne Stroustrup’s initial synthesis of C and Simula ideas. It died immediately following the implementation of its successor, C++.

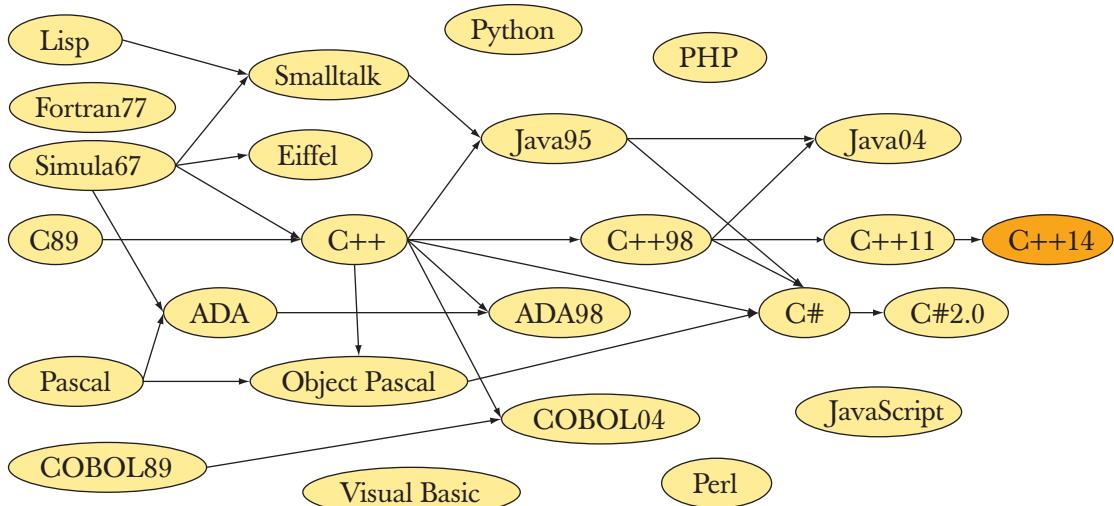
Language discussions often focus on elegance and advanced features. However, C and C++ didn't become two of the most successful languages in the history of computing that way. Their strengths were flexibility, performance, and stability. Major software systems live over decades, often exhaust their hardware resources, and often suffer completely unexpected changes of requirements. C and C++ have been able to thrive in that environment. Our favorite Dennis Ritchie quote is, "Some languages are designed to prove a point; others are designed to solve a problem." By "others," he primarily meant C. Bjarne Stroustrup is fond of saying, "Even I knew how to design a prettier language than C++." The aim for C++ – as for C – was not abstract beauty (though we strongly appreciate that when we can get it), but utility.

## References

- Alexander Stepanov's publications: [www.stepanovpapers.com](http://www.stepanovpapers.com).
- Bjarne Stroustrup's home page: [www.stroustrup.com](http://www.stroustrup.com).
- ISO/IEC 14882:2011. *Programming Languages – C++*. (The C++ standard.)
- Stroustrup, Bjarne. "A History of C++: 1979–1991. Proceedings of the ACM History of Programming Languages Conference (HOPL-2). *ACM SIGPLAN Notices*, Vol. 28 No. 3, 1993.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 978-0321563842.
- Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 978-0321958310.
- Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility." *The C/C++ Users Journal*. July, Aug., and Sept. 2002.
- Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 1991–2006. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

### 22.2.7 Today

What programming languages are currently used and for what? That's a *really* hard question to answer. The family tree of current languages is – even in a most abbreviated form – somewhat crowded and messy:



In fact, most of the statistics we find on the web (and elsewhere) are hardly better than rumors because they measure things that are only weakly correlated with use, such as number of web postings containing the name of a programming language, compiler shipments, academic papers, book sales, etc. All such measures favor the new over the established. Anyway, what is a programmer? Someone who uses a programming language every day? How about a student who writes small programs just to learn? A professor who just talks about programming? A physicist who writes a program almost every year? Is a professional programmer who – almost by definition – uses several programming languages every week counted many times or just once? We have seen each of these questions answered each way for different statistics.

However, we feel obliged to give you an opinion, so in 2014 there are about 10 million professional programmers in the world. For that opinion we rely on IDC (a data-gathering firm), discussions with publishers and compiler suppliers, and various web sources. Feel free to quibble, but we know the number is larger than 1 million and less than 100 million for any halfway reasonable definition of *programmer*. Which language do they use? Ada, C, C++, C#, COBOL, Fortran, Java, PERL, PHP, Python, and Visual Basic probably (just probably) account for significantly more than 90% of all programs.

In addition to the languages mentioned here, we could list dozens or even hundreds more. Apart from trying to be fair to interesting or important languages, we see no point. Please seek out information yourself as needed. A professional

knows several languages and learns new ones as needed. There is no “one true language” for all people and all applications. In fact, all major systems we can think of use more than one language.

### 22.2.8 Information sources

Each individual language description above has a reference list. These are references covering several languages:

#### *More language designer links/photos*

[www.angelfire.com/tx4/cus/people/](http://www.angelfire.com/tx4/cus/people/).

#### *A few examples of languages*

<http://dmoz.org/Computers/Programming/Languages/>.

#### *Textbooks*

Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.

Sebesta, Robert W. *Concepts of Programming Languages*. Addison-Wesley, 2003. ISBN 0321193628.

#### *History books*

Bergin, T.J., and R. G. Gibson, eds. *History of Programming Languages – II*. Addison-Wesley, 1996. ISBN 0201895021.

Hailpern, Brent, and Barbara G. Ryder, eds. Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III). San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.

Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts—The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 978-0465042265.

Sammet, Jean. *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. ISBN 0137299885.

Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.

## Review

1. What are some uses of history?
2. What are some uses of a programming language? List examples.
3. List some fundamental properties of programming languages that are objectively good.
4. What do we mean by abstraction? By higher level of abstraction?

5. What are our four high-level ideals for code?
6. List some potential advantages of high-level programming.
7. What is reuse and what good might it do?
8. What is procedural programming? Give a concrete example.
9. What is data abstraction? Give a concrete example.
10. What is object-oriented programming? Give a concrete example.
11. What is generic programming? Give a concrete example.
12. What is multi-paradigm programming? Give a concrete example.
13. When was the first program run on a stored-program computer?
14. What work made David Wheeler noteworthy?
15. What was the primary contribution of John Backus's first language?
16. What was the first language designed by Grace Murray Hopper?
17. In which field of computer science did John McCarthy primarily work?
18. What were Peter Naur's contributions to Algol60?
19. What work made Edsger Dijkstra noteworthy?
20. What languages did Niklaus Wirth design and implement?
21. What languages did Anders Hejlsberg design?
22. What was Jean Ichbiah's role in the Ada project?
23. What style of programming did Simula pioneer?
24. Where (outside Oslo) did Kristen Nygaard teach?
25. What work made Ole-Johan Dahl noteworthy?
26. Ken Thompson was the main designer of which operating system?
27. What work made Doug McIlroy noteworthy?
28. What is Brian Kernighan's most famous book?
29. Where did Dennis Ritchie work?
30. What work made Bjarne Stroustrup noteworthy?
31. What languages did Alex Stepanov use trying to design the STL?
32. List ten languages not described in §22.2.
33. Scheme is a dialect of which language?
34. What are C++'s two most prominent ancestors?
35. What does the "C" in C++ stand for?
36. Is Fortran an acronym? If so, what for?
37. Is COBOL an acronym? If so, what for?
38. Is Lisp an acronym? If so, what for?
39. Is Pascal an acronym? If so, what for?
40. Is Ada an acronym? If so, what for?
41. Which is the best programming language?

## Terms

In this chapter “Terms” are really languages, people, and organizations.

- Languages:
  - Ada
  - Algol
  - BCPL
  - C
  - C++
  - COBOL
  - Fortran
  - Lisp
  - Pascal
  - Scheme
  - Simula
- Organizations:
  - Bell Laboratories
  - Borland
  - Cambridge University (England)
  - ETH (Swiss Federal Technical University)
  - IBM
  - MIT
  - Norwegian Computer Center
  - Princeton University
  - Stanford University
  - Technical University of Copenhagen
  - U.S. Department of Defense
  - U.S. Navy
- People:
  - Charles Babbage
  - John Backus
  - Ole-Johan Dahl
  - Edsger Dijkstra
  - Anders Hejlsberg
  - Grace Murray Hopper
  - Jean Ichbiah
  - Brian Kernighan
  - John McCarthy
  - Doug McIlroy
  - Peter Naur
  - Kristen Nygaard
  - Dennis Ritchie
  - Alex Stepanov
  - Bjarne Stroustrup
  - Ken Thompson
  - David Wheeler
  - Niklaus Wirth

## Exercises

1. Define *programming*.
2. Define *programming language*.
3. Go through the book and look at the chapter vignettes. Which ones were from computer scientists? Write one paragraph summarizing what each of those scientists contributed.
4. Go through the book and look at the chapter vignettes. Which ones were not from computer scientists? Identify the country of origin and field of work of each.
5. Write a “Hello, World!” program in each of the languages mentioned in this chapter.
6. For each language mentioned in this chapter, look at a popular textbook and see what is used as the first complete program. Write that program in all of the other languages. Warning: This could easily be a 100-program project.
7. We have obviously “missed” many important languages. In particular, we essentially had to cut all developments after C++. Make a list of five modern languages that you think ought to be covered and write a page and a half – along the lines of the language sections in this chapter – on three of those.
8. What is C++ used for and why? Write a 10- to 20-page report.
9. What is C used for and why? Write a 10- to 20-page report.
10. Pick one language (not C or C++) and write a 10- to 20-page description of its origins, aims, and facilities. Give plenty of concrete examples. Who uses it and for what?
11. Who currently holds the Lucasian Chair in Cambridge?
12. Of the language designers mentioned in this chapter, who has a degree in mathematics? Who does not?
13. Of the language designers mentioned in this chapter, who has a Ph.D.? In which field? Who does not have a Ph.D.?
14. Of the language designers mentioned in this chapter, who has received the Turing Award? What is that? Find the actual Turing Award citations for the winners mentioned here.
15. Write a program that, given a file of (name,year) pairs, such as (Algol,1960) and (C,1974), graphs the names on a timeline.
16. Modify the program from the previous exercise so that it reads a file of (name,year,(ancestors)) tuples, such as (Fortran,1956,()), (Algol,1960, (Fortran)), and (C++,1985,(C,Simula)), and graphs them on a timeline with arrows from ancestors to descendants. Use this program to draw improved versions of the diagrams in §22.2.2 and §22.2.7.

## Postscript

Obviously, we have only scratched the surface of both the history of programming languages and of the ideals that fuel the quest for better software. We consider history and ideals sufficiently important to feel really bad about that. We hope to have conveyed some of our excitement and some idea of the immensity of the quest for better software and better programming as it manifests itself through the design and implementation of programming languages. That said, please remember that programming – the development of quality software – is the fundamental and important topic; a programming language is just a tool for that.



## Text Manipulation

“Nothing is so obvious that it’s obvious . . .

The use of the word ‘obvious’ indicates  
the absence of a logical argument.”

—Errol Morris

This chapter is mostly about extracting information from text. We store lots of our knowledge as words in documents, such as books, email messages, or “printed” tables, just to later have to extract it into some form that is more useful for computation. Here, we review the standard library facilities most used in text processing: **strings**, **iostreams**, and **maps**. Then, we introduce regular expressions (**regexs**) as a way of expressing patterns in text. Finally, we show how to use regular expressions to find and extract specific data elements, such as ZIP codes (postal codes), from text and to verify the format of text files.

<b>23.1 Text</b>	<b>23.8 Regular expression syntax</b>
<b>23.2 Strings</b>	<b>23.8.1 Characters and special characters</b>
<b>23.3 I/O streams</b>	<b>23.8.2 Character classes</b>
<b>23.4 Maps</b>	<b>23.8.3 Repeats</b>
<b>23.4.1 Implementation details</b>	<b>23.8.4 Grouping</b>
<b>23.5 A problem</b>	<b>23.8.5 Alternation</b>
<b>23.6 The idea of regular expressions</b>	<b>23.8.6 Character sets and ranges</b>
<b>23.6.1 Raw string literals</b>	<b>23.8.7 Regular expression errors</b>
<b>23.7 Searching with regular expressions</b>	<b>23.9 Matching with regular expressions</b>
	<b>23.10 References</b>

## 23.1 Text

We manipulate text essentially all the time. Our books are full of text, much of what we see on our computer screens is text, and our source code is text. Our communication channels (of all sorts) overflow with words. Everything that is communicated between two humans could be represented as text, but let's not go overboard. Images and sound are usually best represented as images and sound (i.e., just bags of bits), but just about everything else is fair game for program text analysis and transformation.

We have been using `iostreams` and `strings` since Chapter 3, so here, we'll just briefly review those libraries. Maps (§23.4) are particularly useful for text processing, so we present an example of their use for email analysis. After this review, this chapter is concerned with searching for patterns in text using regular expressions (§23.5–10).

## 23.2 Strings

A `string` contains a sequence of characters and provides a few useful operations, such as adding a character to a `string`, giving the length of the `string`, and concatenating `strings`. Actually, the standard `string` provides quite a few operations, but most are useful only when you have to do fairly complicated text manipulation at a low level. Here, we just mention a few of the more useful. You can look up their details (and the full set of `string` operations) in a manual or expert-level textbook should you need them. They are found in `<string>` (note: not `<string.h>`):

Selected <code>string</code> operations	
<code>s1 = s2</code>	Assign <code>s2</code> to <code>s1</code> ; <code>s2</code> can be a <code>string</code> or a C-style string.
<code>s += x</code>	Add <code>x</code> at end; <code>x</code> can be a character, a <code>string</code> , or a C-style string.
<code>s[i]</code>	Subscripting.
<code>s1+s2</code>	Concatenation; the characters in the resulting <code>string</code> will be a copy of those from <code>s1</code> followed by a copy of those from <code>s2</code> .
<code>s1==s2</code>	Comparison of <code>string</code> values; <code>s1</code> or <code>s2</code> , but not both, can be a C-style string. Also <code>!=</code> .
<code>s1&lt;s2</code>	Lexicographical comparison of <code>string</code> values; <code>s1</code> or <code>s2</code> , but not both, can be a C-style string. Also <code>&lt;=</code> , <code>&gt;</code> , and <code>&gt;=</code> .
<code>s.size()</code>	Number of characters in <code>s</code> .
<code>s.length()</code>	Number of characters in <code>s</code> .
<code>s.c_str()</code>	C-style version of characters in <code>s</code> .
<code>s.begin()</code>	Iterator to first character.
<code>s.end()</code>	Iterator to one beyond the end of <code>s</code> .
<code>s.insert(pos,x)</code>	Insert <code>x</code> before <code>s[pos]</code> ; <code>x</code> can be a <code>string</code> or a C-style string. <code>s</code> expands to make room for the characters from <code>x</code> .
<code>s.append(x)</code>	Insert <code>x</code> after the last character of <code>s</code> ; <code>x</code> can be a <code>string</code> or a C-style string. <code>s</code> expands to make room for the characters from <code>x</code> .
<code>s.erase(pos)</code>	Remove trailing characters from <code>s</code> starting with <code>s[pos]</code> . <code>s</code> 's size becomes <code>pos</code> .
<code>s.erase(pos,n)</code>	Remove <code>n</code> characters from <code>s</code> starting at <code>s[pos]</code> . <code>s</code> 's size becomes <code>max(pos,size-n)</code> .
<code>pos = s.find(x)</code>	Find <code>x</code> in <code>s</code> ; <code>x</code> can be a character, a <code>string</code> , or a C-style string; <code>pos</code> is the index of the first character found, or <code>string::npos</code> (a position off the end of <code>s</code> ).
<code>in&gt;&gt;s</code>	Read a whitespace-separated word into <code>s</code> from <code>in</code> .
<code>getline(in,s)</code>	Read a line into <code>s</code> from <code>in</code> .
<code>out&lt;&lt;s</code>	Write from <code>s</code> to <code>out</code> .

The I/O operations are explained in Chapters 10 and 11 and summarized in §23.3. Note that the input operations into a `string` expand the `string` as needed, so that overflow cannot happen.

The `insert()` and `append()` operations may move characters to make room for new characters. The `erase()` operation moves characters “forward” in the `string` to make sure that no gap is left where we erased a character.

The standard library `string` is really a template, called `basic_string`, that supports a variety of character sets, such as Unicode, providing thousands of characters (such as  $\mathcal{L}$ ,  $\Omega$ ,  $\mu$ ,  $\delta$ ,  $\oplus$ , and  $\natural$  in addition to “ordinary characters”). For example, if you have a type holding a Unicode character, such as `Unicode`, you can write

```
basic_string<Unicode> a_unicode_string;
```

The standard string, `string`, which we have been using, is simply the `basic_string` of an ordinary `char`:

```
using string = basic_string<char>; // string means basic_string<char> (§20.5)
```

We do not cover Unicode characters or Unicode strings here, but if you need them you can look them up, and you’ll find that they can be handled (by the language, by `string`, by `iostreams`, and by regular expressions) much as ordinary characters and strings. If you need to use Unicode characters, it is best to ask someone experienced for advice; to be useful, your code has to follow not just the language rules but also some system conventions.

In the context of text processing, it is important that just about anything can be represented as a string of characters. For example, here on this page, the number `12.333` is represented as a string of six characters (surrounded by whitespace). If we read this number, we must convert those characters to a floating-point number before we can do arithmetic operations on the number. This leads to a need to convert values to `strings` and `strings` to values. In §11.4, we saw how to turn an integer into a `string` using an `ostringstream`. This technique can be generalized to any type that has a `<<` operator:

```
template<typename T> string to_string(const T& t)
{
    ostringstream os;
    os << t;
    return os.str();
}
```

For example:

```
string s1 = to_string(12.333);
string s2 = to_string(1+5*6-99/7);
```

The value of `s1` is now `"12.333"` and the value of `s2` is `"17"`. In fact, `to_string()` can be used not just for numeric values, but for any class `T` with a `<<` operator.

The opposite conversion, from `strings` to numeric values, is about as easy, and as useful:

```
struct bad_from_string : std::bad_cast { // class for reporting string cast errors
    const char* what() const override
    {
        return "bad cast from string";
    }
};

template<typename T> T from_string(const string& s)
{
    istringstream is {s};
    T t;
    if (!(is >> t)) throw bad_from_string{};
    return t;
}
```

For example:

```
double d = from_string<double>("12.333");

void do_something(const string& s)
try
{
    int i = from_string<int>(s);
    // ...
}
catch (bad_from_string e) {
    error("bad input string",s);
}
```

The added complication of `from_string()` compared to `to_string()` comes because a `string` can represent values of many types. This implies that we must say which type of value we want to extract from a `string`. It also implies that the `string` we are looking at may not hold a representation of a value of the type we expect. For example:

```
int d = from_string<int>("Mary had a little lamb"); // oops!
```

So there is a possibility of error, which we have represented by the exception `bad_from_string`. In §23.9, we demonstrate how `from_string()` (or an equivalent function) is essential for serious text processing because we need to extract numeric values from text fields. In §16.4.3, we saw how an equivalent function `get_int()` was used in GUI code.

Note how `to_string()` and `from_string()` are similar in function. In fact, they are roughly inverses of each other; that is (ignoring details of whitespace, rounding, etc.), for every “reasonable type `T`” we have

```
s==to_string(from_string<T>(s))      // for all s
```

and

```
t==from_string<T>(to_string(t))      // for all t
```

Here, “reasonable” means that `T` should have a default constructor, a `>>` operator, and a matching `<<` operator defined.

Note also how the implementations of `to_string()` and `from_string()` both use a `stringstream` to do all the hard work. This observation has been used to define a general conversion operation between any two types with matching `<<` and `>>` operations:



```
template<typename Target, typename Source>
Target to(Source arg)
{
    stringstream interpreter;
    Target result;

    if (!(interpreter << arg))                      // write arg into stream
        || !(interpreter >> result)                  // read result from stream
        || !(interpreter >> std::ws.eof())           // stuff left in stream?
        throw runtime_error{"to<>() failed"};

    return result;
}
```

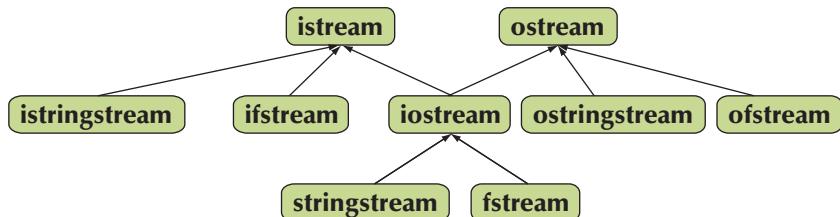
The curious and clever `!(interpreter>>std::ws.eof())` reads any whitespace that might be left in the `stringstream` after we have extracted the result. Whitespace is allowed, but there should be no more characters in the input and we can check that by seeing if we are at “end of file.” So if we try to read an `int` from a `string`, both `to<int>("123")` and `to<int>("123 ")` will succeed, but `to<int>("123.5")` will not because of that last `.5`.

## 23.3 I/O streams

Considering the connection between strings and other types, we get to I/O streams. The I/O stream library doesn't just do input and output; it also performs conversions between string formats and types in memory. The standard library I/O streams provide facilities for reading, writing, and formatting strings of characters. The `iostream` library is described in Chapters 10 and 11, so here we'll just summarize:

Stream I/O	
<code>in &gt;&gt; x</code>	Read from <code>in</code> into <code>x</code> according to <code>x</code> 's type.
<code>out &lt;&lt; x</code>	Write <code>x</code> to <code>out</code> according to <code>x</code> 's type.
<code>in.get(c)</code>	Read a character from <code>in</code> into <code>c</code> .
<code>getline(in,s)</code>	Read a line from <code>in</code> into the string <code>s</code> .

The standard streams are organized into a class hierarchy (§14.3):



Together, these classes supply us with the ability to do I/O to and from files and strings (and anything that can be made to look like a file or a string, such as a keyboard and a screen; see Chapter 10). As described in Chapters 10 and 11, the `iostreams` provide fairly elaborate formatting facilities. The arrows indicate inheritance (see §14.3), so that, for example, a `stringstream` can be used as an `iostream` or as an `istream` or as an `ostream`.

Like `string`, `iostreams` can be used with larger character sets such as Unicode, much like ordinary characters. Please again note that if you need to use Unicode I/O, it is best to ask someone experienced for advice; to be useful, your code has to follow not just the language rules but also some system conventions.

## 23.4 Maps

Associative arrays (maps, hash tables) are key (pun intended) to a lot of text processing. The reason is simply that when we process text, we collect information,

and that information is often associated with text strings, such as names, addresses, postal codes, Social Security numbers, job titles, etc. Even if some of those text strings could be converted into numeric values, it is often more convenient and simpler to treat them as text and use that text for identification. The word-counting example (§21.6) is a good simple example. If you don't feel comfortable using [maps](#), please reread §21.6 before proceeding.

Consider email. We often search and analyze email messages and email logs – usually with the help of some program (e.g., Thunderbird or Outlook). Mostly, those programs save us from seeing the complete source of the messages, but all the information about who sent, who received, where the message went along the way, and much more is presented to the programs as text in a message header. That's a complete message. There are thousands of tools for analyzing the headers. Most use regular expressions (as described in §23.5–9) to extract information and some form of associative arrays to associate related messages. For example, we often search a mail file to collect all messages with the same sender, the same subject, or containing information on a particular topic.

Here, we will use a very simplified mail file to illustrate some of the techniques for extracting data from text files. The headers are real RFC2822 headers from [www.faqs.org/rfcs/rfc2822.html](http://www.faqs.org/rfcs/rfc2822.html). Consider:

```
xxx
xxx
-----
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600
Message-ID: <1234@local.machine.example>

This is a message just to say hello.
So, "Hello".
-----
From: Joe Q. Public <john.q.public@example.com>
To: Mary Smith <@machine.tld:mary@example.net>, jdoe@test .example
Date: Tue, 1 Jul 2003 10:52:37 +0200
Message-ID: <5678.21-Nov-1997@example.com>

Hi everyone.
-----
To: "Mary Smith: Personal Account" <smith@home.example>
From: John Doe <jdoe@machine.example>
Subject: Re: Saying Hello
```

**Date:** Fri, 21 Nov 1997 11:00:00 -0600  
**Message-ID:** <abcd.1234@local.machine.tld>  
**In-Reply-To:** <3456@example.net>  
**References:** <1234@local.machine.example> <3456@example.net>

## This is a reply to your reply.

1

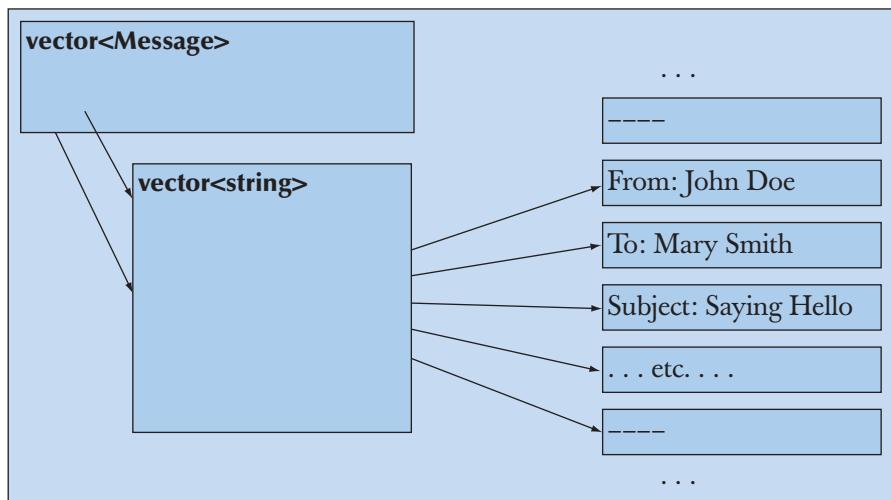
Basically, we have abbreviated the file by throwing most of the information away and eased the analysis by terminating each message by a line containing just ---- (four dashes). We will write a small “toy application” that finds all messages sent by “John Doe” and write out their “Subject.” If we can do that, we can do many interesting things.

First, we must consider whether we want random access to the data or just to analyze it as it streams by in an input stream. We choose the former because in a real program, we would probably be interested in several senders or in several pieces of information from a given sender. Also, it's actually the harder of the two tasks, so it will allow us to examine more techniques. In particular, we get to use iterators again.

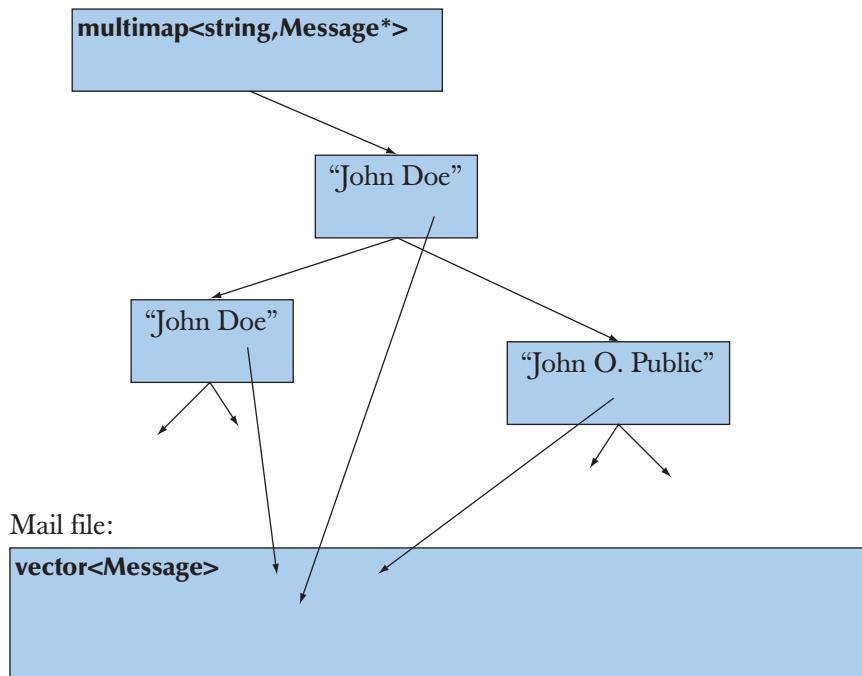


Our basic idea is to read a complete mail file into a structure (which we call a **Mail\_file**). This structure will hold all the lines of the mail file (in a **vector<string>**) and indicators of where each individual message starts and ends (in a **vector<Message>**):

Mail file:



To this, we will add iterators and **begin()** and **end()** functions, so that we can iterate through the lines and through the messages in the usual way. This “boilerplate” will allow us convenient access to the messages. Given that, we will write our “toy application” to gather all the messages from each sender so that they are easy to access together:



Finally, we will write out all the subject headers of messages from “John Doe” to illustrate a use of the access structures we have created.

We use many of the basic standard library facilities:

```

#include<string>
#include<vector>
#include<map>
#include<fstream>
#include<iostream>
using namespace std;
  
```

We define a **Message** as a pair of iterators into a **vector<string>** (our vector of lines):

```

typedef vector<string>::const_iterator Line_iter;

class Message { // a Message points to the first and the last lines of a message
    Line_iter first;
    Line_iter last;
public:
    Message(Line_iter p1, Line_iter p2) :first{p1}, last{p2} {}
    Line_iter begin() const { return first; }
    Line_iter end() const { return last; }
    // ...
};

```

We define a **Mail\_file** as a structure holding lines of text and messages:

```

using Mess_iter = vector<Message>::const_iterator;

struct Mail_file {                                // a Mail_file holds all the lines from a file
// and simplifies access to messages
    string name;                                // file name
    vector<string> lines;                      // the lines in order
    vector<Message> m;                         // Messages in order

    Mail_file(const string& n); // read file n into lines

    Mess_iter begin() const { return m.begin(); }
    Mess_iter end() const { return m.end(); }
};

```

Note how we added iterators to the data structures to make it easy to systematically traverse them. We are not actually going to use standard library algorithms here, but if we wanted to, the iterators are there to allow it.

To find information in a message and extract it, we need two helper functions:

```

// find the name of the sender in a Message;
// return true if found
// if found, place the sender's name in s:
bool find_from_addr(const Message* m, string& s);

// return the subject of the Message, if any, otherwise "";
string find_subject(const Message* m);

```

Finally, we can write some code to extract information from a file:

```
int main()
{
    Mail_file mfile {"my-mail-file.txt"};           // initialize mfile from a file

    // first gather messages from each sender together in a multimap:

    multimap<string, const Message*> sender;

    for (const auto& m : mfile) {
        string s;
        if (find_from_addr(&m,s))
            sender.insert(make_pair(s,&m));
    }

    // now iterate through the multimap
    // and extract the subjects of John Doe's messages:
    auto pp = sender.equal_range("John Doe <jdoe@machine.example>");
    for(auto p = pp.first; p!=pp.second; ++p)
        cout << find_subject(p->second) << '\n';
}
}
```



Let us examine the use of maps in detail. We used a **multimap** (§20.10, §B.4) because we wanted to gather many messages from the same address together in one place. The standard library **multimap** does that (makes it easy to access elements with the same key). Obviously (and typically), we have two parts to our task:

- Build the map.
- Use the map.

We build the **multimap** by traversing all the messages and inserting them into the **multimap** using **insert()**:

```
for (const auto& m : mfile) {
    string s;
    if (find_from_addr(&m,s))
        sender.insert(make_pair(s,&m));
}
```

What goes into a map is a (key,value) pair, which we make with **make\_pair()**. We use our “homemade” **find\_from\_addr()** to find the name of the sender.

Why did we first put the **Messages** in a **vector** and then later build a **multimap**? Why didn't we just put the **Messages** into a **map** immediately? The reason is simple and fundamental:

- First, we build a general structure that we can use for many things.
- Then, we use that for a particular application.

That way, we build up a collection of more or less reusable components. Had we immediately built a **map** in the **Mail\_file**, we would have had to redefine it whenever we wanted to do some different task. In particular, our **multimap** (significantly called **sender**) is sorted based on the Address field of a message. Most other applications would not find that order particularly useful: they might be looking at Return fields, Recipients, Copy-to fields, Subject fields, time stamps, etc.

This way of building applications in stages (or *layers*, as the parts are sometimes called) can dramatically simplify the design, implementation, documentation, and maintenance of programs. The point is that each part does only one thing and does it in a straightforward way. On the other hand, doing everything at once would require cleverness. Obviously, our “extracting information from an email header” program was just a tiny example of an application. The value of keeping separate things separate, modularization, and gradually building an application increases with size.

To extract information, we simply find all the entries with the key **"John Doe"** using the **equal\_range()** function (§B.4.10). Then we iterate through all the elements in the sequence [first,second) returned by **equal\_range()**, extracting the subject by using **find\_subject()**:

```
auto pp = sender.equal_range("John Doe <jdoe@machine.example>");

for (auto p = pp.first; p!=pp.second; ++p)
    cout << find_subject(p->second) << '\n';
```

When we iterate over the elements of a **map**, we get a sequence of (key,value) pairs, and as with all **pairs**, the first element (here, the **string** key) is called **first** and the second (here, the **Message** value) is called **second** (§21.6).

### 23.4.1 Implementation details

Obviously, we need to implement the functions we use. It was tempting to save a tree by leaving this as an exercise, but we decided to make this example complete. The **Mail\_file** constructor opens the file and constructs the **lines** and **m** vectors:

```
Mail_file::Mail_file(const string& n)
// open file named n
// read the lines from n into lines
```

```

// find the messages in the lines and compose them in m
// for simplicity assume every message is ended by a ----- line
{
    ifstream in {n};           // open the file
    if (!in) {
        cerr << "no " << n << '\n';
        exit(1);                // terminate the program
    }

    for (string s; getline(in,s); )   // build the vector of lines
        lines.push_back(s);

    auto first = lines.begin();      // build the vector of Messages
    for (auto p = lines.begin(); p!=lines.end(); ++p) {
        if (*p == "-----") {       // end of message
            m.push_back(Message(first,p));
            first = p+1;           // ----- not part of message
        }
    }
}

```

The error handling is rudimentary. If this were a program we planned to give to friends to use, we'd have to do better.

---

### TRY THIS



We really mean it: do run this example and make sure you understand the result. What would be “better error handling”? Modify `Mail_file`'s constructor to handle likely formatting errors related to the use of `-----`.

The `find_from_addr()` and `find_subject()` functions are simple placeholders until we can do a better job of identifying information in a file (using regular expressions; see §23.6–10):

```

int is_prefix(const string& s, const string& p)
    // is p the first part of s?
{
    int n = p.size();
    if (string(s,0,n)==p) return n;
    return 0;
}

```

```

bool find_from_addr(const Message* m, string& s)
{
    for (const auto& x : m)
        if (int n = is_prefix(x, "From: "))
            s = string(x,n);
            return true;
    }
    return false;
}

string find_subject(const Message* m)
{
    for (const auto& x : m)
        if (int n = is_prefix(x, "Subject: ")) return string(x,n);
    return "";
}

```

Note the way we use substrings: **string(s,n)** constructs a string consisting of the tail of **s** from **s[n]** onward (**s[n]..s[s.size()-1]**), whereas **string(s,0,n)** constructs a string consisting of the characters **s[0]..s[n-1]**. Since these operations actually construct new strings and copy characters, they should be used with care where performance matters.

Why are the **find\_from\_addr()** and **find\_subject()** functions so different? For example, one returns a **bool** and the other a **string**. They are different because we wanted to make a point:

- **find\_from\_addr()** distinguishes between finding an address line with an empty address ("") and finding no address line. In the first case, **find\_from\_addr()** returns **true** (because it found an address) and sets **s** to "" (because the address just happens to be empty). In the second case, it returns **false** (because there was no address line).
- **find\_subject()** returns "" if there was an empty subject or if there was no subject line.

Is the distinction made by **find\_from\_addr()** useful? Necessary? We think that the distinction can be useful and that we definitely should be aware of it. It is a distinction that comes up again and again when looking for information in a data file: did we find the field we were looking for and was there something useful in it? In a real program, both the **find\_from\_addr()** and **find\_subject()** functions would have been written in the style of **find\_from\_addr()** to allow users to make that distinction.

This program is not tuned for performance, but it is probably fast enough for most uses. In particular, it reads its input file only once, and it does not keep

multiple copies of the text from that file. For large files, it may be a good idea to replace the `multimap` with an `unordered_multimap`, but unless you measure, you'll never know.

See §21.6 for an introduction to the standard library associative containers (`map`, `multimap`, `set`, `unordered_map`, and `unordered_multimap`).

## 23.5 A problem

I/O streams and `string` help us read and write sequences of characters, help us store them, and help with basic manipulation. However, it is very common to do operations on text where we need to consider the context of a string or involve many similar strings. Consider a trivial example. Take an email message (a sequence of words) and see if it contains a U.S. state abbreviation and ZIP code (two letters followed by five digits):

```
for (string s; cin>>s; ) {
    if (s.size()==7
        && isalpha(s[0]) && isalpha(s[1])
        && isdigit(s[2]) && isdigit(s[3]) && isdigit(s[4])
        && isdigit(s[5]) && isdigit(s[6]))
        cout << "found " << s << '\n';
}
```

Here, `isalpha(x)` is `true` if `x` is a letter and `isdigit(x)` is `true` if `x` is a digit (see §11.6).

There are several problems with this simple (too simple) solution:

- It's verbose (four lines, eight function calls).
- We miss (intentionally?) every postal code not separated from its context by whitespace (such as `"TX77845"`, `TX77845-1234`, and `ATX77845`).
- We miss (intentionally?) every postal code with a space between the letters and the digits (such as `TX 77845`).
- We accept (intentionally?) every postal code with the letters in lower case (such as `tx77845`).
- If we decide to look for a postal code in a different format (such as `CB3 0FD`), we have to completely rewrite the code.

There has to be a better way! Before revealing that way, let's just consider the problems we would encounter if we decided to stay with the “good old simple way” of writing more code to handle more cases.

- If we want to deal with more than one format, we'd have to start adding **if**-statements or **switch**-statements.
- If we want to deal with upper and lower case, we'd explicitly have to convert (usually to lower case) or add yet another **if**-statement.
- We need to somehow (how?) describe the context of what we want to find. That implies that we must deal with individual characters rather than with strings, and that implies that we lose many of the advantages provided by **iostreams** (§7.8.2).

If you like, you can try to write the code for that, but it is obvious that on this track we are headed for a mess of **if**-statements dealing with a mess of special cases. Even for this simple example, we need to deal with alternatives (e.g., both five- and nine-digit ZIP codes). For many other examples, we need to deal with repetition (e.g., any number of digits followed by an exclamation mark, such as **123!** and **123456!**). Eventually, we would also have to deal with both prefixes and suffixes. As we observed (§11.1–2), people's tastes in output formats are not limited by a programmer's desire for regularity and simplicity. Just think of the bewildering variety of ways people write dates:

**2007-06-05**  
**June 5, 2007**  
**jun 5, 2007**  
**5 June 2007**  
**6/5/2007**  
**5/6/07**  
...

At this point – if not earlier – the experienced programmer declares, “There has to be a better way!” (than writing more ordinary code) and proceeds to look for it. The simplest and most popular solution is using what are called *regular expressions*. Regular expressions are the backbone of much text processing, the basis for the Unix grep command (see exercise 8), and an essential part of languages heavily used for such processing (such as AWK, PERL, and PHP).

The regular expressions we will use are part of the C++ standard library. They are compatible with the regular expressions in PERL. This makes many explanations, tutorials, and manuals available. For example, see the C++ standard committee's working paper (look for “WG21” on the web), John Maddock's **boost::regex** documentation, and most PERL tutorials. Here, we will describe the fundamental concepts and some of the most basic and useful ways of using regular expressions.

---

**TRY THIS**

The last two paragraphs “carelessly” used several names and acronyms without explanation. Do a bit of web browsing to see what we are referring to.

## 23.6 The idea of regular expressions

The basic idea of a regular expression is that it defines a pattern that we can look for in a text. Consider how we might concisely describe the pattern for a simple U.S. postal code, such as **TX77845**. Here is a first attempt:

**wwddddd**

Here, **w** represents “any letter” and **d** represents “any digit.” We use **w** (for “word”) because **l** (for “letter”) is too easily confused with the digit 1. This notation works for this simple example, but let’s try it for the nine-digit ZIP code format (such as **TX77845-5629**). How about

**wwddddd-dddd**



That looks OK, but how come that **d** means “any digit” but **-** means “plain” dash? Somehow, we ought to indicate that **w** and **d** are special: they represent character classes rather than themselves (**w** means “an **a** or a **b** or a **c** or . . . ”) and **d** means “a **1** or a **2** or a **3** or . . . ”). That’s too subtle. Let’s prefix a letter that is a name of a class of characters with a backslash in the way special characters have always been indicated in C++ (e.g., **\n** is newline in a string literal). This way we get

**\w\w\d\d\d\d\d-\d\d\d\d**

This is a bit ugly, but at least it is unambiguous, and the backslashes make it obvious that “something unusual is going on.” Here, we represent repetition of a character by simply repeating. That can be a bit tedious and is potentially error-prone. Quick: Did we really get the five digits before the dash and four after it right? We did, but nowhere did we actually *say* **5** and **4**, so you had to count to make sure. We could add a count after a character to indicate repetition. For example:

**\w2\d5-\d4**

However, we really ought to have some syntax to show that the **2**, **5**, and **4** in that pattern are counts, rather than just the alphanumeric characters **2**, **5**, and **4**. Let's indicate counts by putting them in curly braces:

**\w{2}\d{5}-\d{4}**

That makes **{** special in the same way as **\** (backslash) is special, but that can't be helped and we can deal with that.

So far, so good, but we have to deal with two more messy details: the final four digits in a ZIP code are optional. We somehow have to be able to say that we will accept both **TX77845** and **TX77845-5629**. There are two fundamental ways of expressing that:

**\w{2}\d{5} or \w{2}\d{5}-\d{4}**

and

**\w{2}\d{5} and optionally -\d{4}**

To say that concisely and precisely, we first have to express the idea of grouping (or sub-pattern) to be able to speak about the **\w{2}\d{5}** and **-\d{4}** parts of **\w{2}\d{5}-\d{4}**. Conventionally, we use parentheses to express grouping:

**(\w{2}\d{5})(-\d{4})**

Now we have split the pattern into two sub-patterns, so we just have to say what we want to do with them. As usual, the cost of introducing a new facility is to introduce another special character: **(** is now “special” just like **\** and **{**. Conventionally **|** is used to express “or” (alternatives) and **?** is used to express something conditional (optional), so we might write

**(\w{2}\d{5})|(\w{2}\d{5}-\d{4})**

and

**(\w{2}\d{5})(-\d{4})?**

As with the curly braces in the count notation (e.g., **\w{2}**), we use the question mark (**?**) as a suffix. For example, **(-\d{4})?** means “optionally **-\d{4}**”; that is, we accept four digits preceded by a dash as a suffix. Actually, we are not using the

parentheses around the pattern for the five-digit ZIP code (`\w{2}\d{5}`) for anything, so we could leave them out:

`\w{2}\d{5}(-\d{4})?`

To complete our solution to the problem stated in §23.5, we could add an optional space after the two letters:

`\w{2} ?\d{5}(-\d{4})?`

That “`?`” looks a bit odd, but of course it’s a space character followed by the `?`, indicating that the space character is optional. If we wanted to avoid a space being so unobtrusive that it looks like a bug, we could put it in parentheses:

`\w{2}( )?\d{5}((-)\d{4})?`

If someone considered that still too obscure, we could invent a notation for a whitespace character, such as `\s` (`s` for “space”). That way we could write

`\w{2}\s?\d{5}(-\d{4})?`

But what if someone wrote two spaces after the letters? As defined so far, the pattern would accept `TX77845` and `TX 77845` but not `TX 77845`. That’s a bit subtle. We need to be able to say “zero or more whitespace characters,” so we introduce the suffix `*` to mean “zero or more” and get

`\w{2}\s*\d{5}(-\d{4})?`

This makes sense if you followed every step of the logical progression. This notation for patterns is logical and extremely terse. Also, we didn’t pick our design choices at random: this particular notation is extremely common and popular. For many text-processing tasks, you need to read and write this notation. Yes, it looks a bit as if a cat walked over the keyboard, and yes, typing a single character wrong (even a space) completely changes the meaning, but please just get used to it. We can’t suggest anything dramatically better, and this style of notation has already been wildly popular for more than 30 years since it was first introduced for the Unix grep command – and it wasn’t completely new even then.

### 23.6.1 Raw string literals

Note all of those backslashes in the regular expression patterns. To get a backslash (`\`) into a C++ string literal we have to precede it with a backslash. Consider our postal code pattern:

```
\w{2}\s*\d{5}(-\d{4})?
```

To represent that pattern as a string literal, we have to write

```
"\"\\w{2}\\s*\\d{5}(-\\d{4})?!"
```

Thinking a bit ahead, we realize that many of the patterns we would like to match contain double quotes (""). To get a double quote into a string literal we have to precede it with a backslash. This can quickly become unmanageable. In fact, in real use this “special character problem” gets so annoying that C++ and other languages have introduced the notion of raw string literals to be able to cope with realistic regular expression patterns. In a raw string literal a backslash is simply a backslash character (rather than an escape character) and a double quote is simply a double quote character (rather than an end of string). As a raw string literal our postal code pattern becomes

```
R"(\w{2}\s*\d{5}(-\d{4})?)"
```

The R"( starts the string and )" terminates it, so the 22 characters of the string are

```
\w{2}\s*\d{5}(-\d{4})?
```

not counting the terminating zero.

## 23.7 Searching with regular expressions

Now, we will use the postal code pattern from the previous section to find postal codes in a file. The program defines the pattern and then reads a file line by line, searching for the pattern. If the program finds an occurrence of the pattern in a line, it writes out the line number and what it found:

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

int main()
{
    ifstream in {"file.txt"};                                // input file
    if (!in) cerr << "no file\n";

    regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"};           // postal code pattern
```

```

int lineno = 0;
for (string line; getline(in,line); ) { // read input line into input buffer
    ++lineno;
    smatch matches; // matched strings go here
    if (regex_search(line, matches, pat))
        cout << lineno << ":" << matches[0] << '\n';
}
}

```

This requires a bit of a detailed explanation. We find the standard library regular expressions in `<regex>`. Given that, we can define a pattern `pat`:

```
regex pat {R"(\w{2}\s*\d{5}(-\d{4})?)"; // postal code pattern}
```

A `regex` pattern is a kind of `string`, so we can initialize it with a string. Here, we used a raw string literal. However, a `regex` is not just a `string`, but the somewhat sophisticated mechanism for pattern matching that is created when you initialize a `regex` (or assign to one) is hidden and beyond the scope of this book. However, once we have initialized a `regex` with our pattern for postal codes, we can apply it to each line of our file:

```

smatch matches;
if (regex_search(line, matches, pat))
    cout << lineno << ":" << matches[0] << '\n';

```

The `regex_search(line, matches, pat)` searches the `line` for anything that matches the regular expression stored in `pat`, and if it finds any matches, it stores them in `matches`. Naturally, if no match was found, `regex_search(line, matches, pat)` returns `false`.

The `matches` variable is of type `smatch`. The `s` stands for “sub” or for “string.” Basically, an `smatch` is a vector of sub-matches of type `string`. The first element, here `matches[0]`, is the complete match. We can treat `matches[i]` as a string if `i < matches.size()`. So if – for a given regular expression – the maximum number of sub-patterns is `N`, we find `matches.size() == N+1`.

So, what is a sub-pattern? A good first answer is “Anything in parentheses in the pattern.” Looking at `\w{2}\s*\d{5}(-\d{4})?`, we see the parentheses around the four-digit extension of the ZIP code. That’s the only sub-pattern we see, so we guess (correctly) that `matches.size() == 2`. We also guess that we can easily access those last four digits. For example:

```

for (string line; getline(in,line); ) {
    smatch matches;
    if (regex_search(line, matches, pat)) {

```

```

        cout << lineno << ":" << matches[0] << '\n';
        if (1<matches.size() && matches[1].matched)           // whole match
            cout << "\t: " << matches[1] << '\n';
        }                                                       // sub-match
    }
}

```

Strictly speaking, we didn't have to test `1<matches.size()` because we already had a good look at the pattern, but we felt like being paranoid (because we have been experimenting with a variety of patterns in `pat` and they didn't all have just one sub-pattern). We can ask if a sub-match succeeded by looking at its `matched` member, here `matches[1].matched`. In case you wonder: when `matches[i].matched` is `false`, the unmatched sub-pattern `matches[i]` prints as the empty string. Similarly, a sub-pattern that doesn't exist, such as `matches[17]` for the pattern above, is treated as an unmatched sub-pattern.

We tried this program with a file containing

```

address TX77845
ffff tx 77843 asasasaa
ggg TX3456–23456
howdy
zzz TX23456–3456sss ggg TX33456–1234
cvzcv TX77845–1234 sdsas
xxxTx77845xxx
TX12345–123456

```

and got the output

```

pattern: "\w{2}\s*\d{5}(-\d{4})?"
1: TX77845
2: tx 77843
5: TX23456–3456
   : -3456
6: TX77845–1234
   : -1234
7: Tx77845
8: TX12345–1234
   : -1234

```

Note that we

- Did not get fooled by the ill-formatted “postal code” on the line that starts with `ggg` (what's wrong with that one?)
- Only found the first postal code from the line with `zzz` (we only asked for one per line)

- Found the correct suffixes on lines 5 and 6
- Found the postal code “hidden” among the `xxx`s on line 7
- Found (unfortunately?) the postal code “hidden” in `TX12345–123456`

## 23.8 Regular expression syntax

We have seen a rather basic example of regular expression matching. Now is the time to consider regular expressions (in the form they are used in the `regex` library) a bit more systematically and completely.

*Regular expressions* (“*regexp*s” or “*regex*s”) is basically a little language for expressing patterns of characters. It is a powerful (expressive) and terse language, and as such it can be quite cryptic. After decades of use, there are many subtle features and several dialects. Here, we will just describe a (large and useful) subset of what appears to be the currently most widely used dialect (the PERL one). Should you need more to express what you need to say or to understand the regular expressions of others, go look on the web. Tutorials (of wildly differing quality) and specifications abound.

The library also supports the ECMAScript, POSIX, awk, grep, and egrep notations and a host of search options. This can be extremely useful, especially if you need to match some pattern specified in another language. You can look up those options if you feel the need to go beyond the basic facilities described here. However, remember that “using the most features” is not an aim of good programming. Whenever you can, take pity on the poor maintenance programmer (maybe yourself in a couple of months) who has to read and understand your code: write code that is not unnecessarily clever and avoid obscure features whenever you can.

### 23.8.1 Characters and special characters

A regular expression specifies a pattern that can be used to match characters from a string. By default, a character in a pattern matches itself in a string. For example, the regular expression (pattern) `"abc"` will match the `abc` in `Is there an abc here?`

The real power of regular expressions comes from “special characters” and character combinations that have special meanings in a pattern:

#### Characters with special meaning

- any single character (a “wildcard”)
- [ character class
- { count

### Characters with special meaning (*continued*)

(	begin grouping
)	end grouping
\	next character has a special meaning
*	zero or more
+	one or more
?	optional (zero or one)
	alternative (or)
^	start of line; negation
\$	end of line

For example,

**x.y**

matches any three-character string starting with an **x** and ending with a **y**, such as **xy**, **x3y**, and **xay**, but not **yxy**, **3xy**, and **xy**.

Note that **{...}**, **\***, **+**, and **?** are suffix operators. For example, **\d+** means “one or more decimal digits.”

If you want to use one of the special characters in a pattern, you have to “escape it” using a backslash; for example, in a pattern **+** is the one-or-more operator, but **\+** is a plus sign.

### 23.8.2 Character classes

The most common combinations of characters are represented in a terse form as “special characters”:

#### Special characters for character classes

<b>\d</b>	a decimal digit	<code>[:digit:]</code>
<b>\l</b>	a lowercase character	<code>[:lower:]</code>
<b>\s</b>	a space (space, tab, etc.)	<code>[:space:]</code>
<b>\u</b>	an uppercase character	<code>[:upper:]</code>
<b>\w</b>	a letter (a–z or A–Z) or digit (0–9) or an underscore (_)	<code>[:alnum:]</code>
<b>\D</b>	not <b>\d</b>	<code>[^[:digit:]]</code>

Special characters for character classes ( <i>continued</i> )		
\L	not \l	[^[:lower:]]
\S	not \s	[^[:space:]]
\U	not \u	[^[:upper:]]
\W	not \w	[^[:alnum:]]

Note that an uppercase special character means “not the lowercase version of that special character.” In particular, \W means “not a letter” rather than “an uppercase letter.”

The entries in the third column (e.g., [:digit:]) give an alternative syntax using a longer name.

Like the `string` and `iostream` libraries, the `regex` library can handle large character sets, such as Unicode. As with `string` and `iostream`, we just mention this so that you can look for help and more information should you need it. Dealing with Unicode text manipulation is beyond the scope of this book.

### 23.8.3 Repeats

Repeating patterns are specified by the suffix operators:

Repetition	
{n}	exactly $n$ times
{n,}	$n$ or more times
{n,m}	at least $n$ and at most $m$ times
*	zero or more, that is, {0,}
+	one or more, that is, {1,}
?	optional (zero or one), that is, {0,1}

For example,

`Ax*`

matches an `A` followed by zero or more `x`s, such as

```
A
Ax
Axx
Axxxxxxxxxxxxxxxxxxxxxx
```

If you want at least one occurrence, use `+` rather than `*`. For example,

**Ax+**

matches an **A** followed by one or more **x**s, such as

**Ax**  
**Axx**  
**Axxxxxxxxxxxxxxxxxxxxxx**

but not

**A**

The common case of zero or one occurrence (“optional”) is represented by a question mark. For example,

**\d-?\d**

matches the two digits with an optional dash between them, such as

**1-2**  
**12**

but not

**1--2**

To specify a specific number of occurrences or a specific range of occurrences, use curly braces. For example,

**\w{2}-\d{4,5}**

matches exactly two letters and a dash (`-`) followed by four or five digits, such as

**Ab-1234**  
**XX-54321**  
**22-54321**

but not

**Ab-123**  
**?b-1234**

Yes, digits are `\w` characters.

### 23.8.4 Grouping

To specify a regular expression as a sub-pattern, you group it using parentheses. For example:

`(\d*:)`

This defines a sub-pattern of zero or more digits followed by a colon. A group can be used as part of a more elaborate pattern. For example:

`(\d*:)?(\\d+)`

This specifies an optional and possibly empty sequence of digits followed by a colon followed by a sequence of one or more digits. No wonder people invented a terse and precise way of saying such things!

### 23.8.5 Alternation

The “or” character `(|)` specifies an alternative. For example:

`Subject: (FW:|Re:)?(.*)`

This recognizes an email subject line with an optional `FW:` or `Re:` followed by zero or more characters. For example:

`Subject: FW: Hello, world!`  
`Subject: Re:`  
`Subject: Norwegian Blue`

but not

`SUBJECT: Re: Parrots`  
`Subject FW: No subject!`

An empty alternative is not allowed:

`(|def) // error`

However, we can specify several alternatives at once:

`(bs|Bs|bS|BS)`

### 23.8.6 Character sets and ranges

The special characters provide a shorthand for the most common classes of characters: digits (`\d`); letters, digits, and underscore (`\w`); etc. (§23.7.2). However, it is easy and often useful to define our own. For example:

<code>[\w @]</code>	a word character, a space, or an <code>@</code>
<code>[a-z]</code>	the lowercase characters from <code>a</code> to <code>z</code>
<code>[a-zA-Z]</code>	upper- or lowercase characters from <code>a</code> to <code>z</code>
<code>[Pp]</code>	an upper- or lowercase <code>P</code>
<code>[\w-]</code>	a word character or a dash (plain <code>-</code> means “range”)
<code>[asdfghjkl;:]</code>	the characters on the middle line of a U.S. QWERTY keyboard
<code>[.]</code>	a dot
<code>[.{{\^*+?^\$}]</code>	a character with special meaning in a regular expression

In a character class specification, a `-` (dash) is used to specify a range, such as `[1-3]` (`1`, `2`, or `3`) and `[w-z]` (`w`, `x`, `y`, or `z`). Please use such ranges carefully: not every language has the same letters and not every letter encoding has the same ordering. If you feel the need for any range that isn’t a sub-range of the most common letters and digits of the English alphabet, consult the documentation.

Note that we can use the special characters, such as `\w` (meaning “any word character”), within a character class specification. So, how do we get a backslash (`\`) into a character class? As usual, we “escape it” with a backslash: `\\\`.

When the first character of a character class specification is `^`, that `^` means “negation.” For example:

<code>[^aeiouy]</code>	not an English vowel
<code>[^d]</code>	not a digit
<code>[ ^aeiouy]</code>	a space, a <code>^</code> , or an English vowel

In the last regular expression, the `^` wasn’t the first character after the `[`, so it was just a character, not a negation operator. Regular expressions can be subtle.

An implementation of `regex` also supplies a set of named character classes for use in matching. For example, if you want to match any alphanumeric character (that is, a letter or a digit: `a-z` or `A-Z` or `0-9`), you can do it by the regular expression `[:alnum:]`. Here, `alnum` is the name of a set of characters (the set of alphanumeric characters). A pattern for a nonempty quoted string of alphanumeric characters would be `"[:alnum:]"`. To put that regular expression into an ordinary string literal, we have to escape the quotes:

```
string s {"\" [:alnum:]" + "\""};
```

Furthermore, to put that string literal into a **regex**, we must escape the backslashes:

```
regex s {"\\\"[:alnum:]\"+\\\""};
```

Using a raw string literal is simpler:

```
regex s2 {R"("[:alnum:]"+")"};
```

Prefer raw string literals for patterns containing backslashes or double quotes. That turns out to be most patterns in many applications.

Using regular expressions leads to a lot of notational conventions. Anyway, here is a list of the standard character classes:

Character classes	
<b>alnum</b>	any alphanumeric character
<b>alpha</b>	any alphabetic character
<b>blank</b>	any whitespace character that is not a line separator
<b>ctrl</b>	any control character
<b>d</b>	any decimal digit
<b>digit</b>	any decimal digit
<b>graph</b>	any graphical character
<b>lower</b>	any lowercase character
<b>print</b>	any printable character
<b>punct</b>	any punctuation character
<b>s</b>	any whitespace character
<b>space</b>	any whitespace character
<b>upper</b>	any uppercase character
<b>w</b>	any word character (alphanumeric characters plus the underscore)
<b>xdigit</b>	any hexadecimal digit character

An implementation of **regex** may provide more character classes, but if you decide to use a named class not listed here, be sure to check if it is portable enough for your intended use.

### 23.8.7 Regular expression errors

What happens if we specify an illegal regular expression? Consider:

```
regex pat1 {"(ghi)"};
regex pat2 {"[c-a]"};
// missing alternative
// not a range
```

When we assign a pattern to a regex, the pattern is checked, and if the regular expression matcher can't use it for matching because it's illegal or too complicated, a **bad\_expression** exception is thrown.

Here is a little program that's useful for getting a feel for regular expression matching:

```
#include <regex>
#include <iostream>
#include <string>
#include <fstream>
#include <sstream>
using namespace std;

// accept a pattern and a set of lines from input
// check the pattern and search for lines with that pattern

int main()
{
    regex pattern;

    string pat;
    cout << "enter pattern: ";
    getline(cin,pat);      // read pattern

    try {
        pattern = pat;    // this checks pat
        cout << "pattern: " << pat << '\n';
    }
    catch (bad_expression) {
        cout << pat << " is not a valid regular expression\n";
        exit(1);
    }

    cout << "now enter lines:\n";
    int lineno = 0;

    for (string line; getline(cin,line); ) {
        ++lineno;
        smatch matches;
```

```

if (regex_search(line, matches, pattern)) {
    cout << "line " << lineno << ":" << line << '\n';
    for (int i = 0; i<matches.size(); ++i)
        cout << "\tmatches[" << i << "]: "
            << matches[i] << '\n';
}
else
    cout << "didn't match\n";
}
}

```

---

### TRY THIS



Get the program to run and use it to try out some patterns, such as `abc`, `x.*x`, `(.*)`, `\([^\)]*\)`, and `\w+ \w+ (\Jr.)?`.

## 23.9 Matching with regular expressions

There are two basic uses of regular expressions:



- *Searching* for a string that matches a regular expression in an (arbitrarily long) stream of data – `regex_search()` looks for its pattern as a substring in the stream.
- *Matching* a regular expression against a string (of known size) – `regex_match()` looks for a complete match of its pattern and the string.

The search for ZIP codes in §23.6 was an example of searching. Here, we will examine an example of matching. Consider extracting data from a table like this:

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER I ALT
0A	12	11	23
1A	7	8	15
1B	4	11	15
2A	10	13	23
3A	10	12	22
4A	7	7	14
4B	10	5	15
5A	19	8	27

KLASSE	ANTAL DRENGE	ANTAL PIGER	ELEVER I ALT
6A	10	9	19
6B	9	10	19
7A	7	19	26
7G	3	5	8
7I	7	3	10
8A	10	16	26
9A	12	15	27
0MO	3	2	5
0P1	1	1	2
0P2	0	5	5
10B	4	4	8
10CE	0	1	1
1MO	8	5	13
2CE	8	5	13
3DCE	3	3	6
4MO	4	1	5
6CE	3	4	7
8CE	4	4	8
9CE	4	9	13
REST	5	6	11
Alle klasser	184	202	386

This table (of the number of students in Bjarne Stroustrup's old primary school in 2007) was extracted from a context (a web page) where it looks nice and is fairly typical of the kind of data we need to analyze:

- It has numeric data fields.
- It has character fields with strings meaningful only to people who understand the context of the table. (Here, that point is emphasized by the use of Danish.)
- The character strings include spaces.
- The “fields” of this data are separated by a “separation indicator,” which in this case is a tab character.



We chose this table to be “fairly typical” and “not too difficult,” but note one subtlety we must face: we can’t actually see the difference between spaces and tab characters; we have to leave that problem to our code.

We will illustrate the use of regular expressions to

- Verify that this table is properly laid out (i.e., every row has the right number of fields)
- Verify that the numbers add up (the last line claims to be the sum of the columns above)



If we can do that, we can do just about anything! For example, we could make a new table where the rows with the same initial digit (indicating the year: first grades start with 1) are merged or see if the number of students is increasing or decreasing over the years in question (see exercises 10–11).

To analyze the table, we need two patterns: one for the header line and one for the rest of the lines:

```
regex header {R"(^[\w ]+([\w ]+)*$)"};
regex row {R"^([\w ]+([\d+)(\ \d+)(\ \d+)$)"};
```



Please remember that we praised the regular expression syntax for terseness and utility; we did not praise it for ease of comprehension by novices. In fact, regular expressions have a well-earned reputation for being a “write-only language.” Let us start with the header. Since it does not contain any numeric data, we could just have thrown away that first line, but – to get some practice – let us parse it. It consists of four “word fields” (“alphanumeric fields”) separated by tabs. These fields can contain spaces, so we cannot simply use plain `\w` to specify its characters. Instead, we use `[\w]`, that is, a word character (letter, digit, or underscore) or a space. One or more of those is written `[\w ]+`. We want the first of those at the start of a line, so we get `^[\w ]+`. The “hat” (`^`) means “start of line.” Each of the rest of the fields can be expressed as a tab followed by some words: `( [\w ]+)`. Now we take an arbitrary number of those followed by an end of line: `( [\w ]+)*$`. The dollar sign (`$`) means “end of line.”

Note how we can’t see that the tab characters are really tabs, but in this case they expand in the typesetting to reveal themselves.

Now for the more interesting part of the exercise: the pattern for the lines from which we want to extract the numeric data. The first field is as before: `^[\w ]+`. It is followed by exactly three numeric fields, each preceded by a tab, `( \d+)`, so that we get

```
^[\w ]+([\d+)(\ \d+)(\ \d+)$
```

which, after putting it into a raw string literal, is

```
R"^(w ]+ ( \d+) ( \d+) ( \d+) $)"
```

Now all we have to do is to use those patterns. First we will just validate the table layout:

```
int main()
{
    ifstream in {"table.txt"}; // input file
    if (!in) error("no input file\n");

    string line; // input buffer
    int lineno = 0;

    regex header {R"(w ]+ ( [w ]+)*$)"};
    regex row {R"(w ]+ ( \d+)( \d+)( \d+)$)"}; // header line
                                                // data line

    if (getline(in, line)) { // check header line
        smatch matches;
        if (!regex_match(line, matches, header))
            error("no header");
    }
    while (getline(in, line)) { // check data line
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            error("bad line", to_string(lineno));
    }
}
```

For brevity, we left out the `#includes`. We are checking all the characters on each line, so we use `regex_match()` rather than `regex_search()`. The difference between those two is exactly that `regex_match()` must match every character of its input to succeed, whereas `regex_search()` looks at the input trying to find a substring that matches. Mistakenly typing `regex_match()` when you meant `regex_search()` (or vice versa) can be a most frustrating bug to find. However, both of those functions use their “matches” argument identically.

We can now proceed to verify the data in that table. We keep a sum of the number of pupils in the boys (“drenge”) and girls (“piger”) columns. For each row, we check that last field (“ELEVER IALT”) really is the sum of the first two

fields. The last row (“Alle klasser”) purports to be the sum of the columns above. To check that, we modify `row` to make the text field a sub-match so that we can recognize “Alle klasser”:

```
int main()
{
    ifstream in {"table.txt"};           // input file
    if (!in) error("no input file");

    string line;                      // input buffer
    int lineno = 0;

    regex header {R"^(?<=[\w ]+)([\w ]+)*$)"};           // header line
    regex row {R"^(?<=[\w ]+)([\d]+)( [\d+])( [\d+])$)"}; // data line

    if (getline(in, line)) {           // check header line
        smatch matches;
        if (regex_match(line, matches, header)) {
            error("no header");
        }
    }

    // column totals:
    int boys = 0;
    int girls = 0;

    while (getline(in, line)) {
        ++lineno;
        smatch matches;
        if (!regex_match(line, matches, row))
            cerr << "bad line: " << lineno << '\n';

        if (in.eof()) cout << "at eof\n";

        // check row:
        int curr_boy = from_string<int>(matches[2]);
        int curr_girl = from_string<int>(matches[3]);
        int curr_total = from_string<int>(matches[4]);
        if (curr_boy+curr_girl != curr_total) error("bad row sum\n");

        if (matches[1]==="Alle klasser") {           // last line
            if (curr_boy != boys) error("boys don't add up\n");
            if (curr_girl != girls) error("girls don't add up\n");
        }
    }
}
```

```
        if (!(in>>ws).eof()) error("characters after total line");
        return 0;
    }

    // update totals:
    boys += curr_boy;
    girls += curr_girl;
}

error("didn't find total line");
}
```

The last row is semantically different from the other rows – it is their sum. We recognize it by its label (“Alle klasser”). We decided to accept no more non-whitespace characters after that last one (using the technique from `to<>()`; §23.2) and to give an error if we did not find it.

We used `from_string()` from §23.2 to extract an integer value from the data fields. We had already checked that those fields consisted exclusively of digits so we did not have to check that the `string-to-int` conversion succeeded.

## 23.10 References

Regular expressions are a popular and useful tool. They are available in many programming languages and in many formats. They are supported by an elegant theory based on formal languages and by an efficient implementation technique based on state machines. The full generality of regular expressions, their theory, their implementation, and the use of state machines in general are beyond the scope of this book. However, because these topics are rather standard in computer science curricula and because regular expressions are so popular, it is not hard to find more information (should you need it or just be interested).

For more information, see:

Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called “The Dragon Book”). Addison-Wesley, 2007. ISBN 0321547985.

Cox, Russ. “Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .).” <http://swtch.com/~rsc/regexp/regexp1.html>.

Maddock, J. boost::regex documentation. [www.boost.org/](http://www.boost.org/).

Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.



## Drill

1. Find out if `regex` is shipped as part of your standard library. Hint: Try `std::regex` and `tr1::regex`.
2. Get the little program from §23.7 to work; that may involve figuring out how to set the project and/or command-line options to link to the `regex` library and use the `regex` headers.
3. Use the program from drill 2 to test the patterns from §23.7.

## Review

1. Where do we find “text”?
2. What are the standard library facilities most frequently useful for text analysis?
3. Does `insert()` add before or after its position (or iterator)?
4. What is Unicode?
5. How do you convert to and from a `string` representation (to and from some other type)?
6. What is the difference between `cin>>s` and `getline(cin,s)` assuming `s` is a `string`?
7. List the standard streams.
8. What is the key of a `map`? Give examples of useful key types.
9. How do you iterate over the elements of a `map`?
10. What is the difference between a `map` and a `multimap`? Which useful `map` operation is missing for `multimap`, and why?
11. What operations are required for a forward iterator?
12. What is the difference between an empty field and a nonexistent field? Give two examples.
13. Why do we need an escape character to express regular expressions?
14. How do you get a regular expression into a `regex` variable?
15. What does `\w+\s\d{4}` match? Give three examples. What string literal would you use to initialize a `regex` variable with that pattern?
16. How (in a program) do you find out if a string is a valid regular expression?
17. What does `regex_search()` do?
18. What does `regex_match()` do?
19. How do you represent the character dot (.) in a regular expression?
20. How do you represent the notion of “at least three” in a regular expression?
21. Is `7` a `\w` character? Is `_` (underscore)?
22. What is the notation for an uppercase character?
23. How do you specify your own character set?
24. How do you extract the value of an integer field?

25. How do you represent a floating-point number as a regular expression?
26. How do you extract a floating-point value from a match?
27. What is a sub-match? How do you access one?

## Terms

match	<code>regex_match()</code>	search
<b>multimap</b>	<code>regex_search()</code>	<b>smatch</b>
pattern	regular expression	sub-pattern

## Exercises

1. Get the email file example to run; test it using a larger file of your own creation. Be sure to include messages that are likely to trigger errors, such as messages with two address lines, several messages with the same address and/or same subject, and empty messages. Also test the program with something that simply isn't a message according to that program's specification, such as a large file containing no `-----` lines.
2. Add a **multimap** and have it hold subjects. Let the program take an input string from the keyboard and print out every message with that string as its subject.
3. Modify the email example from §23.4 to use regular expressions to find the subject and sender.
4. Find a real email message file (containing real email messages) and modify the email example to extract subject lines from sender names taken as input from the user.
5. Find a large email message file (thousands of messages) and then time it as written with a **multimap** and with that **multimap** replaced by an **unordered\_multimap**. Note that our application does not take advantage of the ordering of the **multimap**.
6. Write a program that finds dates in a text file. Write out each line containing at least one date in the format **line-number: line**. Start with a regular expression for a simple format, e.g., 12/24/2000, and test the program with that. Then, add more formats.
7. Write a program (similar to the one in the previous exercise) that finds credit card numbers in a file. Do a bit of research to find out what credit card formats are really used.
8. Modify the program from §23.8.7 so that it takes as inputs a pattern and a file name. Its output should be the numbered lines (**line-number: line**) that contain a match of the pattern. If no matches are found, no output should be produced.

9. Using `eof()` (§B.7.2), it is possible to determine which line of a table is the last. Use that to (try to) simplify the table-checking program from §23.9. Be sure to test your program with files that end with empty lines after the table and with files that don't end with a newline at all.
10. Modify the table-checking program from §23.9 to write a new table where the rows with the same initial digit (indicating the year: first grades start with 1) are merged.
11. Modify the table-checking program from §23.9 to see if the number of students is increasing or decreasing over the years in question.
12. Write a program, based on the program that finds lines containing dates (exercise 6), that finds all dates and reformats them to the ISO yyyy-mm-dd format. The program should take an input file and produce an output file that is identical to the input file except for the changed date formatting.
13. Does dot `(.)` match `'\n'`? Write a program to find out.
14. Write a program that, like the one in §23.8.7, can be used to experiment with pattern matching by typing in a pattern. However, have it read a file into memory (representing a line break with the newline character, `'\n'`), so that you can experiment with patterns spanning line breaks. Test it and document a dozen test patterns.
15. Describe a pattern that cannot be expressed as a regular expression.
16. For experts only: Prove that the pattern found in the previous exercise really isn't a regular expression.

## Postscript



It is easy to get trapped into the view that computers and computation are all about numbers, that computing is a form of math. Obviously, it is not. Just look at your computer screen; it is full of text and pictures. Maybe it's busy playing music. For every application, it is important to use proper tools. In the context of C++, that means using appropriate libraries. For text manipulation, the regular expression library is often a key tool – and don't forget the `maps` and the standard algorithms.





# Numerics

“For every complex problem there is an answer that is clear, simple, and wrong.”

—H. L. Mencken

This chapter is an overview of some fundamental language and library facilities supporting numeric computation. We present the basic problems of size, precision, and truncation. The central part of the chapter is a discussion of multidimensional arrays – both C-style and an  $\mathcal{N}$ -dimensional matrix library. We introduce random numbers as frequently needed for testing, simulation, and games. Finally, we list the standard mathematical functions and briefly introduce the basic functionality of the standard library complex numbers.

<b>24.1 Introduction</b>	<b>24.6 An example: solving linear equations</b>
<b>24.2 Size, precision, and overflow</b>	<b>24.6.1 Classical Gaussian elimination</b>
24.2.1 Numeric limits	<b>24.6.2 Pivoting</b>
<b>24.3 Arrays</b>	<b>24.6.3 Testing</b>
<b>24.4 C-style multidimensional arrays</b>	<b>24.7 Random numbers</b>
<b>24.5 The Matrix library</b>	<b>24.8 The standard mathematical functions</b>
24.5.1 Dimensions and access	<b>24.9 Complex numbers</b>
24.5.2 1D Matrix	<b>24.10 References</b>
24.5.3 2D Matrix	
24.5.4 Matrix I/O	
24.5.5 3D Matrix	

## 24.1 Introduction

For some people, numerics – that is, serious numerical computations – are everything. Many scientists, engineers, and statisticians are in this category. For many people, numerics are sometimes essential. A computer scientist occasionally collaborating with a physicist would be in this category. For most people, a need for numerics – beyond simple arithmetic of integers and floating-point numbers – is rare. The purpose of this chapter is to address language-technical details needed to deal with simple numerical problems. We do not attempt to teach numerical analysis or the finer points of floating-point operations; such topics are far beyond the scope of this book and blend with domain-specific topics in the application areas. Here, we present

- Issues related to the built-in types having fixed size, such as precision and overflow
- Arrays, both the built-in notion of multidimensional arrays and a **Matrix** library that is better suited to numerical computation
- A most basic description of random numbers
- The standard library mathematical functions
- Complex numbers

The emphasis is on the **Matrix** library that makes handling of matrices (multidimensional arrays) trivial.

## 24.2 Size, precision, and overflow

When we use the built-in types and usual computational techniques, numbers are stored in fixed amounts of memory; that is, the integer types (**int**, **long**, etc.)

are only approximations of the mathematical notion of integers (whole numbers) and the floating-point types (**float**, **double**, etc.) are (only) approximations of the mathematical notion of real numbers. This implies that from a mathematical point of view, some computations are imprecise or wrong. Consider:

```
float x = 1.0/333;  
float sum = 0;  
for (int i=0; i<333; ++i) sum+=x;  
cout << setprecision(15) << sum << "\n";
```

Running this, we do not get 1 as someone might naively expect, but rather

**0.999999463558197**

We expected something like that. What we see here is an effect of a rounding error. A floating-point number has only a fixed number of bits, so we can always “fool it” by specifying a computation that requires more bits to represent a result than the hardware provides. For example, the rational number  $1/3$  cannot be represented exactly as a decimal number (however many decimals we use). Neither can  $1/333$ , so when we add 333 copies of **x** (the machine’s best approximation of  $1/333$  as a **float**), we get something that is slightly different from 1. Whenever we make significant use of floating-point numbers, rounding errors will occur; the only question is whether the error significantly affects the result.

Always check that your results are plausible. When you compute, you must have some notion of what a reasonable result would look like or you could easily get fooled by some “silly bug” or computation error. Be aware of the possibility of rounding errors and if in doubt, consult an expert or read up on numerical techniques.

---

### TRY THIS



Replace **333** in the example with **10** and run the example again. What result would you expect? What result did you get? You have been warned!

---

The effects of integers being of fixed size can surface more dramatically. The reason is that floating-point numbers are by definition approximations of (real) numbers, so they tend to lose precision (i.e., lose the least significant bits). Integers, on the other hand, tend to overflow (i.e., lose the most significant bits). That tends to make floating-point errors sneaky (and often unnoticed by novices) and integer errors spectacular (and typically hard not to notice). Remember that we prefer errors to manifest themselves early and spectacularly so that we can fix them.



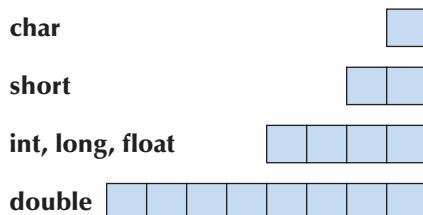
Consider an integer problem:

```
short int y = 40000;
int i = 1000000;
cout << y << " " << i*i << "\n";
```

Running this, we got the output

**-25536 -727379968**

That was expected. What we see here is the effect of overflow. Integer types represent (relatively) small integers only. There just aren't enough bits to exactly represent every number we need in a way that's amenable to efficient computation. Here, a 2-byte **short** integer could not represent 40,000 and a 4-byte **int** can't represent 1,000,000,000,000. The exact sizes of C++ built-in types (§A.8) depend on the hardware and the compiler; **sizeof(x)** gives you the size of **x** in bytes for a variable **x** or a type **x**. By definition, **sizeof(char)==1**. We can illustrate sizes like this:



These sizes are for Windows using a Microsoft compiler. C++ supplies integers and floating-point numbers of a variety of sizes, but unless you have a very good reason for something else, stick to **char**, **int**, and **double**. In most (but of course not all) programs, the remaining integer and floating-point types are more trouble than they are worth.

You can assign an integer to a floating-point variable. If the integer is larger than the floating-point type can represent, you lose precision. For example:

```
cout << "sizes: " << sizeof(int) << ' ' << sizeof(float) << '\n';
int x = 2100000009; // large int
float f = x;
cout << x << ' ' << f << '\n';
cout << setprecision(15) << x << ' ' << f << '\n';
```

On our machine, this produced

**Sizes: 4 4**

```
2100000009 2.1e+009
2100000009 2100000000
```

A **float** and an **int** take up the same amount of space (4 bytes). A **float** is represented as a “mantissa” (typically a value between 0 and 1) and an exponent (mantissa $\times 10^{\text{exponent}}$ ), so it cannot represent exactly the largest **int**. (If we tried to, where would we find space for the mantissa after we had taken the space needed for the exponent?) As it should, **f** represented **2100000009** as approximately correct as it could. However, that last **9** was too much for it to represent exactly – and that was of course why we chose that number.

On the other hand, when you assign a floating-point number to an integer, you get truncation; that is, the fractional part – the digits after the decimal point – is simply thrown away. For example:

```
float f = 2.8;
int x = f;
cout << x << ' ' << f << '\n';
```

The value of **x** will be **2**. It will not be **3** as you might imagine if you are used to “4/5 rounding rules.” C++ **float-to-int** conversions truncate rather than round.

When you calculate, you must be aware of possible overflow and truncation. C++ will not catch such problems for you. Consider:

```
void f(int i, double fpd)
{
    char c = i;           // yes: chars really are very small integers
    short s = i;          // beware: an int may not fit in a short int
    i = i+1;             // what if i was the largest int?
    long lg = i*i;        // beware: a long may not be any larger than an int
    float fps = fpd;      // beware: a large double may not fit in a float
    i = fpd;              // truncates: e.g., 5.7 -> 5
    fps = i;              // you can lose precision (for very large int values)
}

void g()
{
    char ch = 0;
    for (int i = 0; i<500; ++i)
        cout << int(ch++) << '\t';
}
```

If in doubt, check, experiment! Don't just despair and don't just read the documentation. Unless you are experienced, it is easy to misunderstand the highly technical documentation related to numerics.

---

### TRY THIS



Run `g()`. Modify `f()` to print out `c`, `s`, `i`, etc. Test it with a variety of values.



The representation of integers and their conversions will be examined further in §25.5.3. When we can, we prefer to limit ourselves to a few data types. That can help minimize confusion. For example, by not using `float` in a program, but only `double`, we eliminate the possibility of `double`-to-`float` conversion problems. In fact, we prefer to limit our use to `int`, `double`, and `complex` (see §24.9) for computation, `char` for characters, and `bool` for logical entities. We deal with the rest of the arithmetic types only when we have to.

#### 24.2.1 Numeric limits



In `<limits>`, `<climits>`, `<limits.h>`, and `<float.h>`, each C++ implementation specifies properties of the built-in types, so that programmers can use those properties to check against limits, set sentinels, etc. These values are listed in §B.9.1 and can be critically important to low-level tool builders. If you think you need them, you are probably too close to the hardware, but there are other uses. For example, it is not uncommon to be curious about aspects of the language implementation, such as "How big is an `int`?" or "Are `chars` signed?" Trying to find the definite and correct answers in the system documentation can be difficult, and the standard only specifies minimum requirements. However, a program giving the answer is trivial to write:

```
cout << "number of bytes in an int: " << sizeof(int) << '\n';
cout << "largest int: " << INT_MAX << '\n';
cout << "smallest int value: " << numeric_limits<int>::min() << '\n';

if (numeric_limits<char>::is_signed)
    cout << "char is signed\n";
else
    cout << "char is unsigned\n";

char ch = numeric_limits<char>::min();      // smallest positive value
cout << "the char with the smallest positive value: " << ch << '\n';
cout << "the int value of the char with the smallest positive value: "
    << int(ch) << '\n';
```

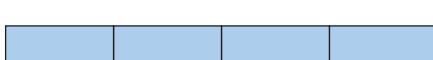
When you write code intended to run on several kinds of hardware, it occasionally becomes immensely valuable to have this kind of information available to the program. The alternative would typically be to hand-code the answers into the program, thereby creating a maintenance hazard.

These limits can also be useful when you want to detect overflow.

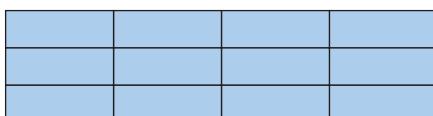
## 24.3 Arrays

An *array* is a sequence of elements where we can access an element by its index (position). Another word for that general notion is *vector*. Here we are particularly concerned with arrays where the elements are themselves arrays: multidimensional arrays. A common word for a multidimensional array is *matrix*. The variety of names is a sign of the popularity and utility of the general concept. The standard **vector** (§B.4), **array** (§20.9), and the built-in array (§A.8.2) are one-dimensional. So, what if we need two dimensions (e.g., a matrix)? If we need seven dimensions?

We can visualize one- and two-dimensional arrays like this:



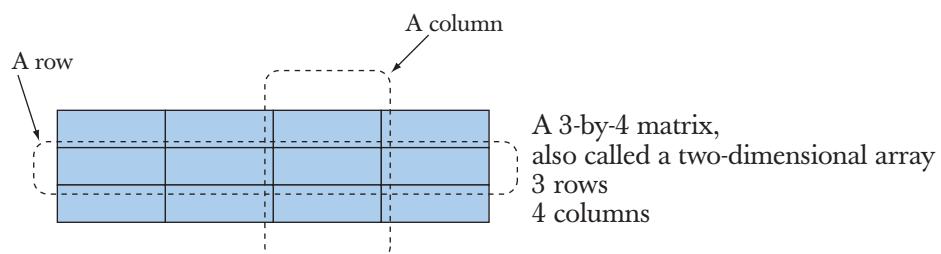
A vector (e.g., **Matrix<int> v(4)**),  
also called a one-dimensional array,  
or even a 1-by- $N$  matrix



A 3-by-4 matrix (e.g., **Matrix<int,2> m(3,4)**),  
also called a two-dimensional array

Arrays are fundamental to most computing (“number crunching”). Most interesting scientific, engineering, statistics, and financial computations rely heavily on arrays.

We often refer to an array as consisting of rows and columns:



A column is a sequence of elements with the same first ( $x$ ) coordinate. A row is a set of elements with the same second ( $y$ ) coordinate.

## 24.4 C-style multidimensional arrays

The C++ built-in array can be used as a multidimensional array. We simply treat a multidimensional array as an array of arrays, that is, an array with arrays as elements. For example:

```
int ai[4];           // 1-dimensional array
double ad[3][4];    // 2-dimensional array
char ac[3][4][5];   // 3-dimensional array
ai[1] = 7;
ad[2][3] = 7.2;
ac[2][3][4] = 'c';
```

This approach inherits the virtues and the disadvantages of the one-dimensional array:

- Advantages
  - Direct mapping to hardware
  - Efficient for low-level operations
  - Direct language support
- Problems
  - C-style multidimensional arrays are arrays of arrays (see below).
  - Fixed sizes (i.e., fixed at compile time). If you want to determine a size at run time, you'll have to use the free store.
  - Can't be passed cleanly. An array turns into a pointer to its first element at the slightest provocation.
  - No range checking. As usual, an array doesn't know its own size.
  - No array operations, not even assignment (copy).

Built-in arrays are widely used for numeric computation. They are also a *major* source of bugs and complexity. For most people, they are a serious pain to write and debug. Look them up if you are forced to use them (e.g., *The C++ Programming Language*). Unfortunately, C++ shares its multidimensional arrays with C, so there is a lot of code “out there” that uses them.

The most fundamental problem is that you can't pass multidimensional arrays cleanly, so you have to fall back on pointers and explicit calculation of locations in a multidimensional array. For example:

```
void f1(int a[3][5]);           // useful for [3][5] matrices only

void f2(int [ ][5], int dim1);   // 1st dimension can be a variable

void f3(int [5][ ], int dim2);   // error: 2nd dimension cannot be a variable

void f4(int[ ][ ], int dim1, int dim2); // error (and wouldn't work anyway)

void f5(int* m, int dim1, int dim2) // odd, but works
{
    for (int i=0; i<dim1; ++i)
        for (int j = 0; j<dim2; ++j) m[i*dim2+j] = 0;
}
```

Here, we pass **m** as an **int\*** even though it is a two-dimensional array. As long as the second dimension needs to be a variable (a parameter), there really isn't any way of telling the compiler that **m** is a (**dim1, dim2**) array, so we just pass a pointer to the start of the memory that holds it. The expression **m[i\*dim2+j]** really means **m[i,j]**, but because the compiler doesn't know that **m** is a two-dimensional array, we have to calculate the position of **m[i,j]** in memory.

This is too complicated, primitive, and error-prone for our taste. It can also be slow because calculating the location of an element explicitly complicates optimization. Instead of trying to teach you all about it, we will concentrate on a C++ library that eliminates the problems with the built-in arrays.

## 24.5 The Matrix library

What are the basic “things” we want from an array/matrix aimed at numerical computation?

- “My code should look very much like what I find in my math/engineering textbook text about arrays.”
  - Or about vectors, matrices, tensors.
- Compile-time and run-time checked.
  - Arrays of any dimension.
  - Arrays with any number of elements in a dimension.

- Arrays are proper variables/objects.
  - You can pass them around.
- Usual array operations:
  - Subscripting: `()`
  - Slicing: `[]`
  - Assignment: `=`
  - Scaling operations (`+=`, `-=`, `*=`, `%=`, etc.)
  - Fused vector operations (e.g., `res[i] = a[i]*c+b[2]`)
  - Dot product (`res` = sum of `a[i]*b[i]`; also known as the `inner_product`)
- Basically, transforms conventional array/vector notation into the code you would laboriously have had to write yourself (and runs at least as efficiently as that).
- You can extend it yourself as needed (no “magic” was used in its implementation).

The `Matrix` library does that and only that. If you want more, such as advanced array functions, sparse arrays, control over memory layout, etc., you must write it yourself or (preferably) use a library that better approximates your needs. However, many such needs can be served by building algorithm and data structures on top of `Matrix`. The `Matrix` library is not part of the ISO C++ standard library. You find it on the book support site as `Matrix.h`. It defines its facilities in namespace `Numeric_lib`. We chose the name “matrix” because “vector” and “array” are even more overused in C++ libraries. The plural of *matrix* is *matrices* (with *matrixes* as a rarer form). Where `Matrix` refers to a C++ language entity, we will use `Matrixes` as the plural to avoid confusion. The implementation of the `Matrix` library uses advanced techniques and will not be described here.

### 24.5.1 Dimensions and access

Consider a simple example:

```
#include "Matrix.h"
using namespace Numeric_lib;

void f(int n1, int n2, int n3)
{
    Matrix<double,1> ad1(n1);           // elements are doubles; one dimension
    Matrix<int,1> ai1(n1);              // elements are ints; one dimension
    ad1(7) = 0;                          // subscript using () — Fortran style
    ad1[7] = 8;                          // [ ] also works — C style
```

```

Matrix<double,2> ad2(n1,n2);
Matrix<double,3> ad3(n1,n2,n3);
ad2(3,4) = 7.5;           // 2-dimensional
ad3(3,4,5) = 9.2;        // 3-dimensional
                            // true multidimensional subscripting
}

```

So, when you define a **Matrix** (an object of a **Matrix** class), you specify the element type and the number of dimensions. Obviously, **Matrix** is a template, and the element type and the number of dimensions are template parameters. The result of giving a pair of arguments to **Matrix** (e.g., **Matrix<double,2>**) is a type (a class) of which you can define objects by supplying arguments (e.g., **Matrix<double,2> ad2(n1,n2)**); those arguments specify the dimensions. So, **ad2** is a two-dimensional array with dimensions **n1** and **n2**, also known as an **n1**-by-**n2** matrix. To get an element of the declared element type from a one-dimensional **Matrix**, you subscript with one index; to get an element of the declared element type from a two-dimensional **Matrix**, you subscript with two indices; and so on.

Like built-in arrays, and **vectors**, our **Matrix** indices are zero-based (rather than 1-based like Fortran arrays); that is, the elements of a **Matrix** are numbered [0,max), where max is the number of elements.

This is simple and “straight out of the textbook.” If you have problems with this, you need to look at an appropriate math textbook, not a programmer’s manual. The only “cleverness” here is that you can leave out the number of dimensions for a **Matrix**: “one-dimensional” is the default. Note also that we can use **[]** for subscripting (C and C++ style) or **( )** for subscripting (Fortran style). Having both allows us to better deal with multiple dimensions. The **[x]** subscript notation always takes a single subscript, yielding the appropriate row of the **Matrix**; if **a** is an  $N$ -dimensional **Matrix**, **a[x]** is an  $N-1$ -dimensional **Matrix**. The **(x,y,z)** subscript notation takes one or more subscripts, yielding the appropriate element of the **Matrix**; the number of subscripts must equal the number of dimensions.

Let’s see what happens when we make mistakes:

```

void f(int n1, int n2, int n3)
{
    Matrix<int,0> ai0;      // error: no 0D matrices

    Matrix<double,1> ad1(5);
    Matrix<int,1> ai(5);
    Matrix<double,1> ad11(7);

    ad1(7) = 0;            // Matrix_error exception (7 is out of range)
    ad1 = ai;              // error: different element types
    ad1 = ad11;            // Matrix_error exception (different dimensions)
}

```

```

Matrix<double,2> ad2(n1); // error: length of 2nd dimension missing
ad2(3) = 7.5; // error: wrong number of subscripts
ad2(1,2,3) = 7.5; // error: wrong number of subscripts

Matrix<double,3> ad3(n1,n2,n3);
Matrix<double,3> ad33(n1,n2,n3);
ad3 = ad33; // OK: same element type, same dimensions
}

```

We catch mismatches between the declared number of dimensions and their use at compile time. Range errors we catch at run time and throw a **Matrix\_error** exception.

The first dimension is the row and the second the column, so we index a 2D matrix (two-dimensional array) with (row,column). We can also use the [row] [column] notation because subscripting a 2D matrix with a single index gives the 1D matrix that is the row. We can visualize that like this:

				a[1][2]
a[0]:	00	01	02	03
a[1]:	10	11	12	13
a[2]:	20	21	22	23

This **Matrix** will be laid out in memory in “row-first” order:

00	01	02	03	10	11	12	13	20	21	22	23
----	----	----	----	----	----	----	----	----	----	----	----

A **Matrix** “knows” its dimensions, so we can address the elements of a **Matrix** passed as an argument very simply:

```

void init(Matrix<int,2>& a) // initialize each element to a characteristic value
{
    for (int i=0; i<a.dim1(); ++i)
        for (int j = 0; j<a.dim2(); ++j)
            a(i,j) = 10*i+j;
}

void print(const Matrix<int,2>& a) // print the elements row by row
{
    for (int i=0; i<a.dim1(); ++i) {
        for (int j = 0; j<a.dim2(); ++j)
            cout << a(i,j) << '\t';
    }
}

```

```

    cout << '\n';
}
}
```

So, `dim1()` is the number of elements in the first dimension, `dim2()` the number of elements in the second dimension, and so on. The type of the elements and the number of dimensions are part of the `Matrix` type, so we cannot write a function that takes any `Matrix` as an argument (but we could write a template to do that):

```
void init(Matrix& a); // error: element type and number of dimensions missing
```

Note that the `Matrix` library doesn't supply matrix operations, such as adding two 4D `Matrix`es or multiplying a 2D `Matrix` with a 1D `Matrix`. Doing so elegantly and efficiently is currently beyond the scope of this library. Matrix libraries of a variety of designs could be built on top of the `Matrix` library (see exercise 12).

### 24.5.2 1D Matrix

What can we do to the simplest `Matrix`, the 1D (one-dimensional) `Matrix`?

We can leave the number of dimensions out of a declaration because 1D is the default:

```
Matrix<int,1> a1(8); // a1 is a 1D Matrix of ints
Matrix<int> a(8); // means Matrix<int,1> a(8);
```

So, `a` and `a1` are of the same type (`Matrix<int,1>`). We can ask for the size (the number of elements) and the dimension (the number of elements in a dimension). For a 1D `Matrix`, those are the same.

```
a.size(); // number of elements in Matrix
a.dim1(); // number of elements in 1st dimension
```

We can ask for the elements as laid out in memory, that is, a pointer to the first element:

```
int* p = a.data(); // extract data as a pointer to an array
```

This is useful for passing `Matrix` data to C-style functions taking pointer arguments. We can subscript:

```
a(i); // ith element (Fortran style), but range checked
a[i]; // ith element (C style), range checked
a(1,2); // error: a is a 1D Matrix
```



It is common for algorithms to refer to part of a **Matrix**. Such a “part” is called a **slice()** (a sub-**Matrix** or a range of elements) and we provide two versions:

```
a.slice(i);      // the elements from a[i] to the last
a.slice(i,n);   // the n elements from a[i] to a[i+n-1]
```

Subscripts and slices can be used on the left-hand side of an assignment as well as on the right. They refer to the elements of their **Matrix** without making copies of them. For example:

```
a.slice(4,4) = a.slice(0,4); // assign first half of a to second half
```

For example, if **a** starts out as

```
{ 1 2 3 4 5 6 7 8 }
```

we get

```
{ 1 2 3 4 1 2 3 4 }
```

Note that the most common slices are the “initial elements” of a **Matrix** and the “last elements”; that is, **a.slice(0,j)** is the range **[0:j]** and **a.slice(j)** is the range **[j:a.size()**). In particular, the example above is most easily written

```
a.slice(4) = a.slice(0,4); // assign first half of a to second half
```

That is, the notation favors the common cases. You can specify **i** and **n** so that **a.slice(i,n)** is outside the range of **a**. However, the resulting slice will refer only to the elements actually in **a**. For example, **a.slice(i,a.size())** refers to the range **[i:a.size())**, and **a.slice(a.size())** and **a.slice(a.size(),2)** are empty **Matrixes**. This happens to be a useful convention for many algorithms. We borrowed that convention from math. Obviously, **a.slice(i,0)** is an empty **Matrix**. We wouldn’t write that deliberately, but there are algorithms that are simpler if **a.slice(i,n)** where **n** happens to be **0** is an empty **Matrix** (rather than an error we have to avoid).



We have the usual (for C++ objects) copy operations that copy all elements:

```
Matrix<int> a2 = a; // copy initialization
a = a2;              // copy assignment
```



We can apply a built-in operation to each element of a **Matrix**:

```
a *= 7;           // scaling: a[i]*=7 for each i (also +=, -=, /=, etc.)
a = 7;            // a[i]=7 for each i
```

This works for every assignment and every composite assignment operator (`=`, `+=`, `-=`, `/=`, `*=`, `%=`, `^=`, `&=`, `|=`, `>>=`, `<<=`) provided the element type supports that operator. We can also apply a function to each element of a **Matrix**:

```
a.apply(f);           // a[i]=f(a[i]) for each element a[i]
a.apply(f,7);         // a[i]=f(a[i],7) for each element a[i]
```

The composite assignment operators and **apply()** modify the elements of their **Matrix** argument. If we instead want to create a new **Matrix** as the result, we can use

```
b = apply(abs,a);   // make a new Matrix with b(i)==abs(a(i))
```

This **abs** is the standard library's absolute value function (§24.8). Basically, **apply(f,x)** relates to **x.apply(f)** as **+** relates to **+=**. For example:

```
b = a*7;            // b[i] = a[i]*7 for each i
a *= 7;             // a[i] = a[i]*7 for each i
y = apply(f,x);     // y[i] = f(x[i]) for each i
x.apply(f);         // x[i] = f(x[i]) for each i
```

Here we get **a==b** and **x==y**.

In Fortran, this second **apply** is called a “broadcast” function and is typically written **f(x)** rather than **apply(f,x)**. To make this facility available for every function **f** (rather than just a selected few functions as in Fortran), we need a name for the “broadcast” operation, so we (re)use **apply**.

In addition, to match the two-argument version of the member **apply**, **a.apply(f,x)**, we provide

```
b = apply(f,a,x);    // b[i]=f(a[i],x) for each i
```

For example:

```
double scale(double d, double s) { return d*s; }
b = apply(scale,a,7);      // b[i] = a[i]*7 for each i
```

Note that the “freestanding” **apply()** takes a function that produces a result from its argument; **apply()** then uses that result to initialize the resulting **Matrix**. Typically it does not modify the **Matrix** to which it is applied. The member **apply()** differs in that it takes a function that modifies its argument; that is, it modifies elements of the **Matrix** to which it is applied. For example:

```
void scale_in_place(double& d, double s) { d *= s; }
b.apply(scale_in_place,7); // b[i] *= 7 for each i
```

We also supply a couple of the most useful functions from traditional numerics libraries:

```
Matrix<int> a3 = scale_and_add(a,8,a2);      // fused multiply and add
int r = dot_product(a3,a);                      // dot product
```

The `scale_and_add()` operation is often referred to as *fused multiply-add* or simply *fma*; its definition is `result(i)=arg1(i)*arg2+arg3(i)` for each `i` in the `Matrix`. The dot product is also known as the `inner_product` and is described in §21.5.3; its definition is `result+=arg1(i)*arg2(i)` for each `i` in the `Matrix` where `result` starts out as `0`.

One-dimensional arrays are very common; you can represent one as a built-in array, a `vector`, or a `Matrix`. You use `Matrix` if you need the matrix operations provided, such as `*=`, or if the `Matrix` has to interact with higher-dimensional `Matrixes`.

You can explain the utility of a library like this as “It matches the math better” or “It saves you from writing all those loops to do things for each element.” Either way, the resulting code is significantly shorter and there are fewer opportunities to make mistakes writing it. The `Matrix` operations – such as copy, assignment to all elements, and operations on all elements – save us from reading or writing a loop (and from wondering if we got the loop exactly right).

`Matrix` supports two constructors for copying data from a built-in array into a `Matrix`. For example:

```
void some_function(double* p, int n)
{
    double val[] = { 1.2, 2.3, 3.4, 4.5 };
    Matrix<double> data(p,n);
    Matrix<double> constants(val);
    // ...
}
```

These are often useful when we have our data delivered in terms of arrays or `vectors` from parts of a program not using `Matrixes`.

Note that the compiler is able to deduce the number of elements of an initialized array, so we don’t have to give the number of elements when we define `constants` – it is `4`. On the other hand, the compiler doesn’t know the number of elements given only a pointer, so for `data` we have to specify both the pointer (`p`) and the number of elements (`n`).

### 24.5.3 2D Matrix

The general idea of the `Matrix` library is that `Matrixes` of different dimensions really are quite similar, except where you need to be specific about dimensions, so most of what we said about a 1D `Matrix` applies to a 2D `Matrix`:

```
Matrix<int,2> a(3,4);

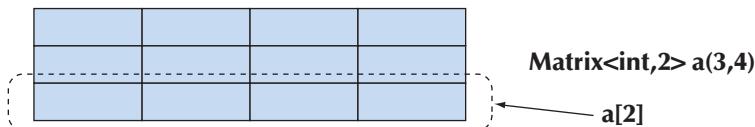
int s = a.size();           // number of elements
int d1 = a.dim0();         // number of elements in a row
int d2 = a.dim1();         // number of elements in a column
int* p = a.data();          // extract data as a pointer to a C-style array
```

We can ask for the total number of elements and the number of elements of each dimension. We can get a pointer to the elements as they are laid out in memory as a matrix.

We can subscript:

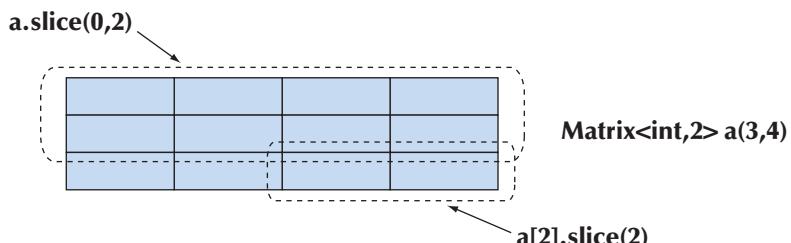
<b>a(i,j);</b>	// (i,j)th element (Fortran style), but range checked
<b>a[i];</b>	// ith row (C style), range checked
<b>a[i][j];</b>	// (i,j)th element (C style)

For a 2D **Matrix**, subscripting with **[i]** yields the 1D **Matrix** that is the **i**th row. This means that we can extract rows and pass them to operations and functions that require a 1D **Matrix** or even a built-in array (**a[i].data()**). Note that **a(i,j)** may be faster than **a[i][j]**, though that will depend a lot on the compiler and optimizer.



We can take slices:

<b>a.slice(i);</b>	// the rows from the a[i] to the last
<b>a.slice(i,n);</b>	// the rows from the a[i] to the a[i+n-1]



Note that a slice of a 2D **Matrix** is itself a 2D **Matrix** (possibly with fewer rows).

The distributed operations are the same as for 1D **Matrixes**. These operations don't care how we organize the elements; they just apply to all elements in the order those elements are laid down in memory:

```
Matrix<int,2> a2 = a;           // copy initialization
a = a2;                      // copy assignment
a *= 7;                      // scaling (and +=, -=, /=, etc.)
a.apply(f);                   // a(i,j)=f(a(i,j)) for each element a(i,j)
a.apply(f,7);                 // a(i,j)=f(a(i,j),7) for each element a(i,j)
b=apply(f,a);                // make a new Matrix with b(i,j)==f(a(i,j))
b=apply(f,a,7);              // make a new Matrix with b(i,j)==f(a(i,j),7)
```

It turns out that swapping rows is often useful, so we supply that:

```
a.swap_rows(1,2);           // swap rows a[1] <-> a[2]
```

There is no **swap\_columns()**. If you need it, write it yourself (exercise 11). Because of the row-first layout, rows and columns are not completely symmetrical concepts. This asymmetry also shows up in that **[i]** yields a row (and we have not provided a column selection operator). In that **(i,j)**, the first index, **i**, selects the row. The asymmetry also reflects deep mathematical properties.

There seems to be an infinite number of “things” that are two-dimensional and thus obvious candidates for applications of 2D **Matrixes**:

```
enum Piece { none, pawn, knight, queen, king, bishop, rook };
Matrix<Piece,2> board(8,8);           // a chessboard

const int white_start_row = 0;
const int black_start_row = 7;

Matrix<Piece> start_row
    = {rook, knight, bishop, queen, king, bishop, knight, rook};

Matrix<Piece> clear_row(8) ;           // 8 elements of the default value
```

The initialization of **clear\_row** takes advantage of **none==0** and that elements are by default initialized to **0**.

We can use **start\_row** and **clear\_row** like this:

```
board[white_start_row] = start_row;           // reset white pieces
for (int i = 1; i<7; ++i) board[i] = clear_row; // clear middle of the board
board[black_start_row] = start_row;             // reset black pieces
```

Note when we extract a row, using **[i]**, we get an lvalue (§4.3); that is, we can assign to the result of **board[i]**.

#### 24.5.4 Matrix I/O

The `Matrix` library provides *very* simple I/O for 1D and 2D `Matrixes`:

```
Matrix<double> a(4);
cin >> a;
cout << a;
```

This will read four whitespace-separated `doubles` delimited by curly braces; for example:

```
{ 1.2 3.4 5.6 7.8 }
```

The output is very similar, so that you can read in what you wrote out.

The I/O for 2D `Matrixes` simply reads and writes a curly-brace-delimited sequence of 1D `Matrixes`. For example:

```
Matrix<int,2> m(2,2);
cin >> m;
cout << m;
```

This will read

```
{
{ 1 2 }
{ 3 4 }
}
```

The output will be very similar.

The `Matrix <<` and `>>` operators are provided primarily to make the writing of simple programs simple. For more advanced uses, it is likely that you will need to replace them with your own. Consequently, the `Matrix <<` and `>>` are placed in the `MatrixIO.h` header (rather than in `Matrix.h`) so that you don't have to include it to use `Matrixes`.

#### 24.5.5 3D Matrix

Basically, a 3D (and higher-dimension) `Matrix` is just like a 2D `Matrix`, except with more dimensions. Consider:

```
Matrix<int,3> a(10,20,30);

a.size();           // number of elements
a.dim1();          // number of elements in dimension 1
a.dim2();          // number of elements in dimension 2
```

```

a.dim3();           // number of elements in dimension 3
int* p = a.data(); // extract data as a pointer to a C-style array
a(i,j,k);         // (i,j,k)th element (Fortran style), but range checked
a[i];             // ith row (C style), range checked
a[i][j][k];       // (i,j,k)th element (C style)
a.slice(i);        // the rows from the ith to the last
a.slice(i,j);     // the rows from the ith to the jth
Matrix<int,3> a2 = a; // copy initialization
a = a2;            // copy assignment
a *= 7;            // scaling (and +=, -=, /=, etc.)
a.apply(f);        // a(i,j,k)=f(a(i,j,k)) for each element a(i,j,k)
a.apply(f,7);      // a(i,j,k)=f(a(i,j,k),7) for each element a(i,j,k)
b=apply(f,a);     // make a new Matrix with b(i,j,k)==f(a(i,j,k))
b=apply(f,a,7);   // make a new Matrix with b(i,j,k)==f(a(i,j,k),7)
a.swap_rows(7,9); // swap rows a[7] <-> a[9]

```

If you understand 2D **Matrixes**, you understand 3D **Matrixes**. For example, here **a** is 3D, so **a[i]** is 2D (provided **i** is in range), **a[i][j]** is 1D (provided **j** is in range), and **a[i][j][k]** is the **int** element (provided **k** is in range).

We tend to see the world as three-dimensional. That leads to obvious uses of 3D **Matrixes** in modeling (e.g., a physics simulation using a Cartesian grid):

```

int grid_nx;          // grid resolution; set at startup
int grid_ny;
int grid_nz;
Matrix<double,3> cube(grid_nx, grid_ny, grid_nz);

```

And then if we add time as a fourth dimension, we get a 4D space needing a 4D **Matrix**. And so on.

For a more advanced version of **Matrix**, supporting general  $\mathcal{N}$ -dimensional matrices, see Chapter 29 of *The C++ Programming Language*.

## 24.6 An example: solving linear equations

 The code for a numerical computation makes sense if you understand the math that it expresses and tends to appear to be utter nonsense if you don't. The example used here should be rather trivial if you have learned basic linear algebra; if not, just see it as an example of transcribing a textbook solution into code with minimal rewording.

The example here is chosen to demonstrate a reasonably realistic and important use of **Matrixes**. We will solve a set (any set) of linear equations of this form:

$$\begin{aligned} a_{1,1}x_1 + \cdots + a_{1,n}x_n &= b_1 \\ &\vdots \\ a_{n,1}x_1 + \cdots + a_{n,n}x_n &= b_n \end{aligned}$$

Here, the  $x$ s designate the  $n$  unknowns;  $as$  and  $bs$  are given constants. For simplicity, we assume that the unknowns and the constants are floating-point values. The goal is to find values for the unknowns that simultaneously satisfy the  $n$  equations. These equations can compactly be expressed in terms of a matrix and two vectors:

$$\mathbf{Ax} = \mathbf{b}$$

Here,  $\mathbf{A}$  is the square  $n$ -by- $n$  matrix defined by the coefficients:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ \vdots & \vdots & \vdots \\ a_{n,1} & \cdots & a_{n,n} \end{bmatrix}$$

The vectors  $\mathbf{x}$  and  $\mathbf{b}$  are the vectors of unknowns and constants, respectively:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \quad \text{and} \quad \mathbf{b} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

This system may have zero, one, or an infinite number of solutions, depending on the coefficients of the matrix  $\mathbf{A}$  and the vector  $\mathbf{b}$ . There are various methods for solving linear systems. We use a classic scheme, called Gaussian elimination (see Freeman and Phillips, *Parallel Numerical Algorithms*; Stewart, *Matrix Algorithms, Volume I*; and Wood, *Introduction to Numerical Analysis*). First, we transform  $\mathbf{A}$  and  $\mathbf{b}$  so that  $\mathbf{A}$  is an upper-triangular matrix. By upper-triangular, we mean all the coefficients below the diagonal of  $\mathbf{A}$  are zero. In other words, the system looks like this:

$$\begin{bmatrix} a_{1,1} & \cdots & a_{1,n} \\ 0 & \ddots & \vdots \\ 0 & 0 & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_n \end{bmatrix}$$

This is easily done. A zero for position  $a(i,j)$  is obtained by multiplying the equation for row  $i$  by a constant so that  $a(i,j)$  equals another element in column  $j$ , say

$a(k,j)$ . That done, we just subtract the two equations and  $a(i,j) = 0$  and the other values in row  $i$  change appropriately.

If we can get all the diagonal coefficients to be nonzero, then the system has a unique solution, which can be found by “back substitution.” The last equation is easily solved:

$$a_{n,n}x_n = b_n$$

Obviously,  $x[n]$  is  $b[n]/a(n,n)$ . That done, eliminate row  $n$  from the system and proceed to find the value of  $x[n-1]$ , and so on, until the value for  $x[1]$  is computed. For each  $n$ , we divide by  $a(n,n)$  so the diagonal values must be nonzero. If that does not hold, the back substitution method fails, meaning that the system has zero or an infinite number of solutions.

### 24.6.1 Classical Gaussian elimination

Now let us look at the C++ code to express this. First, we’ll simplify our notation by conventionally naming the two **Matrix** types that we are going to use:

```
typedef Numeric_lib::Matrix<double, 2> Matrix;
typedef Numeric_lib::Matrix<double, 1> Vector;
```

Next we will express our desired computation:

```
Vector classical_gaussian_elimination(Matrix A, Vector b)
{
    classical_elimination(A, b);
    return back_substitution(A, b);
}
```

That is, we make copies of our inputs **A** and **b** (using call by value), call a function to solve the system, and then calculate the result to return by back substitution. The point is that our breakdown of the problem and our notation for the solution are right out of the textbook. To complete our solution, we have to implement **classical\_elimination()** and **back\_substitution()**. Again, the solution is in the textbook:

```
void classical_elimination(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    // traverse from 1st column to the next-to-last
    // filling zeros into all elements under the diagonal:
```

```

for (Index j = 0; j<n-1; ++j) {
    const double pivot = A(j,j);
    if (pivot == 0) throw Elim_failure(j);

    // fill zeros into each element under the diagonal of the ith row:
    for (Index i = j+1; i<n; ++i) {
        const double mult = A(i,j) / pivot;
        A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
        b(i) -= mult*b(j);      // make the corresponding change to b
    }
}
}

```

The “pivot” is the element that lies on the diagonal of the row we are currently dealing with. It must be nonzero because we need to divide by it; if it is zero we give up by throwing an exception:

```

Vector back_substitution(const Matrix& A, const Vector& b)
{
    const Index n = A.dim1();
    Vector x(n);

    for (Index i = n-1; i>= 0; --i) {
        double s = b(i)-dot_product(A[i].slice(i+1),x.slice(i+1));

        if (double m = A(i,i))
            x(i) = s/m;
        else
            throw Back_subst_failure(i);
    }

    return x;
}

```

## 24.6.2 Pivoting

We can avoid the divide-by-zero problem and also achieve a more robust solution by sorting the rows to get zeros and small values away from the diagonal. By “more robust” we mean less sensitive to rounding errors. However, the values change as we go along placing zeros under the diagonal, so we have to also

reorder to get small values away from the diagonal (that is, we can't just reorder the matrix and then use the classical algorithm):

```
void elim_with_partial_pivot(Matrix& A, Vector& b)
{
    const Index n = A.dim1();

    for (Index j = 0; j<n; ++j) {
        Index pivot_row = j;

        // look for a suitable pivot:
        for (Index k = j+1; k<n; ++k)
            if (abs(A(k,j)) > abs(A(pivot_row,j))) pivot_row = k;

        // swap the rows if we found a better pivot:
        if (pivot_row!=j) {
            A.swap_rows(j,pivot_row);
            std::swap(b(j), b(pivot_row));
        }

        // elimination:
        for (Index i = j+1; i<n; ++i) {
            const double pivot = A(j,i);
            if (pivot==0) error("can't solve: pivot==0");
            const double mult = A(i,j)/pivot;
            A[i].slice(j) = scale_and_add(A[j].slice(j), -mult, A[i].slice(j));
            b(i) -= mult*b(j);
        }
    }
}
```

We use **swap\_rows()** and **scale\_and\_multiply()** to make the code more conventional and to save us from writing an explicit loop.

### 24.6.3 Testing

Obviously, we have to test our code. Fortunately, there is a simple way to do that:

```
void solve_random_system(Index n)
{
    Matrix A = random_matrix(n);           // see §24.7
    Vector b = random_vector(n);
```

```

cout << "A = " << A << '\n';
cout << "b = " << b << '\n';

try {
    Vector x = classical_gaussian_elimination(A, b);
    cout << "classical elim solution is x = " << x << '\n';
    Vector v = A*x;
    cout << " A*x = " << v << '\n';
}
catch(const exception& e) {
    cerr << e.what() << '\n';
}

```

We can get to the **catch** clause in three ways:

- A bug in the code (but, being optimists, we don't think there are any)
- An input that trips up **classical\_elimination** (**elim\_with\_partial\_pivot** could do better in many cases)
- Rounding errors

However, our test is not as realistic as we'd like because genuinely random matrices are unlikely to cause problems for **classical\_elimination**.

To verify our solution, we print out **A\*x**, which had better equal **b** (or close enough for our purpose, given rounding errors). The likelihood of rounding errors is the reason we didn't just do

```
if (A*x!=b) error("substitution failed");
```

Because floating-point numbers are just approximations to real numbers, we have to accept approximately correct answers. In general, using **==** and **!=** on the result of a floating-point computation is best avoided: floating point is inherently an approximation.

The **Matrix** library doesn't define multiplication of a matrix with a vector, so we did that for this program:

```

Vector operator*(const Matrix& m, const Vector& u)
{
    const Index n = m.dim1();
    Vector v(n);
    for (Index i = 0; i<n; ++i) v(i) = dot_product(m[i],u);
    return v;
}
```

Again, a simple `Matrix` operation did most of the work for us. The `Matrix` output operations came from `MatrixIO.h` as described in §24.5.4. The `random_matrix()` and `random_vector()` functions are simple uses of random numbers (§24.7) and are left as an exercise. `Index` is a type alias (§A.16) for the index type used by the `Matrix` library. We brought it into scope with a `using` declaration:

```
using Numeric_lib::Index;
```

## 24.7 Random numbers

If you ask people for a random number, most say 7 or 17, so it has been suggested that those are the “most random” numbers. People essentially never give the answer 0. Zero is seen to be such a nice round number that it is not perceived as “random” and could therefore be deemed the “least random” number. From a mathematical point of view this is utter nonsense: it is not an individual number that is random. What we often need, and what we often refer to as random numbers, is a sequence of numbers that conform to some distribution and where you cannot easily predict the next number in the sequence given the previous ones. Such numbers are most useful in testing (that’s one way of generating a lot of test cases), in games (that is one way of making sure that the next run of the game differs from the previous run), and in simulations (we can make a simulated entity behave in a “random” fashion within the limits of its parameters).

As a practical tool and a mathematical problem, random numbers reach a high degree of sophistication to match their real-world importance. Here, we will just touch the basics as needed for simple testing and simulation. In `<random>`, the standard library provides a sophisticated set of facilities for generating random numbers to match a variety of mathematical distributions. The standard library random number facilities are based on two fundamental notions:

- *Engines* (random number engines): An engine is a function object that generates a uniformly distributed sequence of integer values.
  - *Distributions*: A distribution is a function object that generates a sequence of values according to a mathematical formula given a sequence of values from an engine as inputs.

For example, consider the function `random_vector()` that was used in §24.6.3. A call `random_vector(n)` produces a `Matrix<double,1>` with `n` elements of type `double` with random values in the range `[0:n]`:

```

for (Index i = 0; i < n; ++i)
    v(i) = ureal(ran);

return v;
}

```

The default engine (**default\_random\_engine**) is simple, cheap to run, and good enough for casual use. For more professional uses, the standard library offers a variety of engines with better randomness properties and different running costs. Examples are **linear\_congurential\_engine**, **mersenne\_twister\_engine**, and **random\_device**. If you want to use those, and in general if you need to do better than the **default\_random\_engine**, you have a bit of reading to do. To get an idea of the quality of your system's random number generator, do exercise 10.

The two random number generators from **std\_lib\_facilities.h** were defined as

```

int randint(int min, int max)
{
    static default_random_engine ran;
    return uniform_int_distribution<>{min,max}(ran);
}

int randint(int max)
{
    return randint(0,max);
}

```

These simple functions can be most useful, but just to try something else, let us generate a normal distribution:

```
auto gen = bind(normal_distribution<double>{15,4.0},
                default_random_engine{});
```

The standard library function **bind()** from **<functional>** constructs a function object that when invoked calls its first argument with its second as the argument. So here, **gen()** returns values according to the normal distribution with its mean at **15** and a variance of **4.0** using the **default\_random\_engine**. We could use it like this:

```
vector<int> hist(2*15);

for (int i = 0; i < 500; ++i)                                // generate histogram of 500 values
    ++hist[int(round(gen()))];
```

```
for (int i = 0; i != hist.size(); ++i) {      // write out histogram
    cout << i << '\t';
    for (int j = 0; j != hist[i]; ++j)
        cout << '*';
    cout << '\n';
}
```

We got

```
0
1
2
3 **
4 *
5 *****
6 ****
7 ****
8 ******
9 *********
10 *****
11 *****
12 *****
13 *****
14 *****
15 *****
16 *****
17 *****
18 *****
19 *****
20 *****
21 *****
22 *****
23 *****
24 *****
25 *
26 *
27
28
29
```

The normal distribution is very common and also known as the Gaussian distribution or (for obvious reasons) simply “the bell curve.” Other distributions include **bernoulli\_distribution**, **exponential\_distribution**, and **chi\_squared\_distribution**. You can find them described in *The C++ Programming Language*. Integer distributions return values in a closed interval **[a:b]**, whereas real (floating-point) distributions return values in open intervals **[a:b)**.

By default, an engine (except possibly **random\_device**) gives the same sequence each time a program is run. That is most convenient for initial debugging. If we want different sequences from an engine, we need to initialize it with different values. Such initializers are conventionally called “seeds.” For example:

```
auto gen1 = bind(uniform_int_distribution<>{0,9},  
                 default_random_engine{});  
auto gen2 = bind(uniform_int_distribution<>{0,9},  
                 default_random_engine{10});  
auto gen3 = bind(uniform_int_distribution<>{0,9},  
                 default_random_engine{5});
```

To get an unpredictable sequence, people often use the time of day (down to the last nanosecond; §26.6.1) or something like that as the seed.

## 24.8 The standard mathematical functions

The standard mathematical functions (**cos**, **sin**, **log**, etc.) are provided by the standard library. Their declarations are found in **<cmath>**.

Standard mathematical functions	
<b>abs(x)</b>	absolute value
<b>ceil(x)</b>	smallest integer <b>&gt;= x</b>
<b>floor(x)</b>	largest integer <b>&lt;= x</b>
<b>sqrt(x)</b>	square root; <b>x</b> must be nonnegative
<b>cos(x)</b>	cosine
<b>sin(x)</b>	sine
<b>tan(x)</b>	tangent
<b>acos(x)</b>	arccosine; result is nonnegative

### Standard mathematical functions (*continued*)

<b>asin(x)</b>	arcsine; result nearest to 0 returned
<b>atan(x)</b>	arctangent
<b>sinh(x)</b>	hyperbolic sine
<b>cosh(x)</b>	hyperbolic cosine
<b>tanh(x)</b>	hyperbolic tangent
<b>exp(x)</b>	base-e exponential
<b>log(x)</b>	natural logarithm, base-e; <b>x</b> must be positive
<b>log10(x)</b>	base-10 logarithm

The standard mathematical functions are provided for types **float**, **double**, **long double**, and **complex** (§24.9) arguments. If you do floating-point computations, you'll find these functions useful. If you need more details, documentation is widely available; your online documentation would be a good place to start.

If a standard mathematical function cannot produce a mathematically valid result, it sets the variable **errno**. For example:

```
errno = 0;
double s2 = sqrt(-1);
if (errno) cerr << "something went wrong with something somewhere";
if (errno == EDOM)      // domain error
    cerr << "sqrt() not defined for negative argument";
pow(very_large,2);      // not a good idea
if (errno==ERANGE)      // range error
    cerr << "pow(" << very_large << ",2) too large for a double";
```

If you do serious mathematical computations, you must check **errno** to ensure that it is still **0** after you get your result. If not, something went wrong. Look at your manual or online documentation to see which mathematical functions can set **errno** and which values they use for **errno**.

As indicated in the example, a nonzero **errno** simply means “Something went wrong.” It is not uncommon for functions not in the standard library to set **errno** in case of error, so you have to look more carefully at the value of **errno** to get an idea of exactly what went wrong. If you test **errno** immediately after a standard library function and if you made sure that **errno==0** before calling it, you can rely on the values as we did with **EDOM** and **ERANGE** in the example. **EDOM** is set for a domain error (i.e., a problem with the result). **ERANGE** is set for a range error (i.e., a problem with the arguments).

Error handling based on `errno` is somewhat primitive. It dates from the first (1975 vintage) C mathematical functions.

## 24.9 Complex numbers

Complex numbers are widely used in scientific and engineering computations. We assume that if you need them, you will know about their mathematical properties, so we'll just show you how complex numbers are expressed in the ISO C++ standard library. You find the declaration of complex numbers and their associated standard mathematical functions in `<complex>`:

```
template<class Scalar> class complex {
    // a complex is a pair of scalar values, basically a coordinate pair
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i) :re(r), im(i) {}
    constexpr complex(const Scalar & r) :re(r), im(Scalar ()) {}
    complex() :re(Scalar()), im(Scalar()) {}

    constexpr Scalar real() { return re; }           // real part
    constexpr Scalar imag() { return im; }          // imaginary part

    // operators: = += -= *= /=
};
```

The standard library `complex` is guaranteed to be supported for scalar types `float`, `double`, and `long double`. In addition to the members of `complex` and the standard mathematical functions (§24.8), `<complex>` offers a host of useful operations:

Complex operators	
<code>z1+z2</code>	addition
<code>z1-z2</code>	subtraction
<code>z1*z2</code>	multiplication
<code>z1/z2</code>	division
<code>z1==z2</code>	equality
<code>z1!=z2</code>	inequality
<code>norm(z)</code>	the square of <code>abs(z)</code>

### Complex operators (continued)

<b>conj(z)</b>	conjugate: if <b>z</b> is <b>{re,im}</b> , then <b>conj(z)</b> is <b>(re,-im)</b>
<b>polar(rho,theta)</b>	make a complex given polar coordinates <b>(rho,theta)</b>
<b>real(z)</b>	real part
<b>imag(z)</b>	imaginary part
<b>abs(z)</b>	also known as rho
<b>arg(z)</b>	also known as theta
<b>out &lt;&lt; z</b>	complex output
<b>in &gt;&gt; z</b>	complex input

Note: **complex** does not provide **<** or **%**.

Use **complex<T>** exactly like a built-in type, such as **double**. For example:

**Using cmplx = complex<double>; // sometimes complex<double> gets verbose**

```
void f(cmplx z, vector<cmplx>& vc)
{
    cmplx z2 = pow(z,2);
    cmplx z3 = z2*9.3+vc[3];
    cmplx sum = accumulate(vc.begin(), vc.end(), cmplx{});
    // ...
}
```

Remember that not all operations that we are used to from **int** and **double** are defined for a **complex**. For example:

```
if (z2<z3)      // error: there is no < for complex numbers
```

Note that the representation (layout) of the C++ standard library complex numbers is compatible with their corresponding types in C and Fortran.

## 24.10 References

Basically, the issues discussed in this chapter, such as rounding errors, **Matrix** operations, and complex arithmetic, are of no interest and make no sense in isolation. We simply describe (some of) the support provided by C++ to people with the need for and knowledge of mathematical concepts and techniques to do numerical computations.

In case you are a bit rusty in those areas or simply curious, we can recommend some information sources:

The MacTutor History of Mathematics archive, <http://www-gap.dcs.st-and.ac.uk/~history>

- A great link for anyone who likes math or simply needs to use math
- A great link for someone who would like to see the human side of mathematics; for example, who is the only major mathematician to win an Olympic medal?
  - Famous mathematicians: biographies, accomplishments
  - Curio
- Famous curves
- Famous problems
- Mathematical topics
  - Algebra
  - Analysis
  - Numbers and number theory
  - Geometry and topology
  - Mathematical physics
  - Mathematical astronomy
  - The history of mathematics
  - . . .

Freeman, T. L., and Chris Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992.

Gullberg, Jan. *Mathematics – From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X. One of the most enjoyable books on basic and useful mathematics. A (rare) math book that you can read for pleasure and also use to look up specific topics, such as matrices.

Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1998. ISBN 0201896842.

Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.

Wood, Alistair. *Introduction to Numerical Analysis*. Addison-Wesley, 1999. ISBN 020194291X.



## Drill

1. Print the size of a `char`, a `short`, an `int`, a `long`, a `float`, a `double`, an `int*`, and a `double*` (use `sizeof`, not `<limits>`).
2. Print out the size as reported by `sizeof` of `Matrix<int> a(10)`, `Matrix<int> b(100)`, `Matrix<double> c(10)`, `Matrix<int,2> d(10,10)`, `Matrix<int,3> e(10,10,10)`.
3. Print out the number of elements of each of the `Matrix`es from 2.
4. Write a program that takes `ints` from `cin` and outputs the `sqrt()` of each `int`, or “no square root” if `sqrt(x)` is illegal for some `x` (i.e., check your `sqrt()` return values).
5. Read ten floating-point values from input and put them into a `Matrix<double>`. `Matrix` has no `push_back()` so be careful to handle an attempt to enter a wrong number of `doubles`. Print out the `Matrix`.
6. Compute a multiplication table for  $[0,n] \times [0,m]$  and represent it as a 2D `Matrix`. Take `n` and `m` from `cin` and print out the table nicely (assume that `m` is small enough that the results fit on a line).
7. Read ten `complex<double>`s from `cin` (yes, `cin` supports `>>` for `complex`) and put them into a `Matrix`. Calculate and output the sum of the ten complex numbers.
8. Read six `ints` into a `Matrix<int,2> m(2,3)` and print them out.

## Review

1. Who uses numerics?
2. What is precision?
3. What is overflow?
4. What is a common size of a `double`? Of an `int`?
5. How can you detect overflow?
6. Where do you find numeric limits, such as the largest `int`?
7. What is an array? A row? A column?
8. What is a C-style multidimensional array?
9. What are the desirable properties of language support (e.g., a library) for matrix computation?
10. What is a dimension of a matrix?
11. How many dimensions can a matrix have (in theory/math)?
12. What is a slice?
13. What is a broadcast operation? List a few.
14. What is the difference between Fortran-style and C-style subscripting?
15. How do you apply an operation to each element of a matrix? Give examples.

16. What is a fused operation?
17. Define *dot product*.
18. What is linear algebra?
19. What is Gaussian elimination?
20. What is a pivot? (In linear algebra? In “real life”?)
21. What makes a number random?
22. What is a uniform distribution?
23. Where do you find the standard mathematical functions? For which argument types are they defined?
24. What is the imaginary part of a complex number?
25. What is the square root of  $-1$ ?

## Terms

array	Fortran	scaling
C	fused operation	size
column	imaginary	<b>sizeof</b>
complex number	<b>Matrix</b>	slicing
dimension	multidimensional	subscripting
dot product	random number	uniform distribution
element-wise operation	real	
<b>errno</b>	row	

## Exercises

1. The function arguments **f** for **a.apply(f)** and **apply(f,a)** are different. Write a **triple()** function for each and use each to triple the elements of an array **{ 1 2 3 4 5 }**. Define a single **triple()** function that can be used for both **a.apply(triple)** and **apply(triple,a)**. Explain why it could be a bad idea to write every function to be used by **apply()** that way.
2. Do exercise 1 again, but with function objects, rather than functions. Hint: **Matrix.h** contains examples.
3. Expert level only (this cannot be done with the facilities described in this book): Write an **apply(f,a)** that can take a **void (T&)**, a **T (const T&)**, and their function object equivalents. Hint: **Boost::bind**.
4. Get the Gaussian elimination program to work; that is, complete it, get it to compile, and test it with a simple example.
5. Try the Gaussian elimination program with **A=={ {0 1} {1 0} }** and **b=={ 5 6 }** and watch it fail. Then, try **elim\_with\_partial\_pivot()**.
6. In the Gaussian elimination example, replace the vector operations **dot\_product()** and **scale\_and\_add()** with loops. Test, and comment on the clarity of the code.

7. Rewrite the Gaussian elimination program without using the **Matrix** library; that is, use built-in arrays or **vectors** instead of **Matrixes**.
8. Animate the Gaussian elimination.
9. Rewrite the nonmember **apply()** functions to return a **Matrix** of the return type of the function applied; that is, **apply(f,a)** should return a **Matrix<R>** where **R** is the return type of **f**. Warning: The solution requires information about templates not available in this book.
10. How random is your **default\_random\_engine**? Write a program that takes two integers **n** and **d** as inputs and calls **randint(n)** **d** times, recording the result. Output the number of draws for each of **[0:n)** and “eyeball” how similar the counts are. Try with low values for **n** and with low values for **d** to see if drawing only a few random numbers causes obvious biases.
11. Write a **swap\_columns()** to match **swap\_rows()** from §24.5.3. Obviously, to do that you have to read and understand some of the existing **Matrix** library code. Don’t worry too much about efficiency: it is not possible to get **swap\_columns()** to run as fast as **swap\_rows()**.
12. Implement

```
Matrix<double> operator*(Matrix<double,2>&,Matrix<double>&);
```

and

```
Matrix<double,N>operator+(Matrix<double,N>&,Matrix<double,N>&)
```

If you need to, look up the mathematical definitions in a textbook.

## Postscript

If you don’t feel comfortable with mathematics, you probably didn’t like this chapter and you’ll probably choose a field of work where you are unlikely to need the information presented here. On the other hand, if you do like mathematics, we hope that you appreciate how closely the fundamental concepts of mathematics can be represented in code.



# Embedded Systems Programming

“‘Unsafe’ means ‘Somebody may die.’”

—Safety officer

We present a view of embedded systems programming; that is, we discuss topics primarily related to writing programs for “gadgets” that do not look like conventional computers with screens and keyboards. We focus on the principles, programming techniques, language facilities, and coding standards needed to work “close to the hardware.” The main language issues addressed are resource management, memory management, pointer and array use, and bit manipulation. The emphasis is on safe use and on alternatives to the use of the lowest-level features. We do not attempt to present specialized machine architectures or direct access to hardware devices; that is what specialized literature and manuals are for. As an example, we present the implementation of an encryption/decryption algorithm.

<b>25.1 Embedded systems</b>	<b>25.5 Bits, bytes, and words</b>
<b>25.2 Basic concepts</b>	<b>25.5.1 Bits and bit operations</b>
25.2.1 Predictability	25.5.2 <b>bitset</b>
25.2.2 Ideals	25.5.3 Signed and unsigned
25.2.3 Living with failure	25.5.4 Bit manipulation
<b>25.3 Memory management</b>	25.5.5 Bitfields
25.3.1 Free-store problems	25.5.6 An example: simple encryption
25.3.2 Alternatives to the general free store	
25.3.3 Pool example	<b>25.6 Coding standards</b>
25.3.4 Stack example	25.6.1 What should a coding standard be?
<b>25.4 Addresses, pointers, and arrays</b>	25.6.2 Sample rules
25.4.1 Unchecked conversions	25.6.3 Real coding standards
25.4.2 A problem: dysfunctional interfaces	
25.4.3 A solution: an interface class	
25.4.4 Inheritance and containers	

## 25.1 Embedded systems

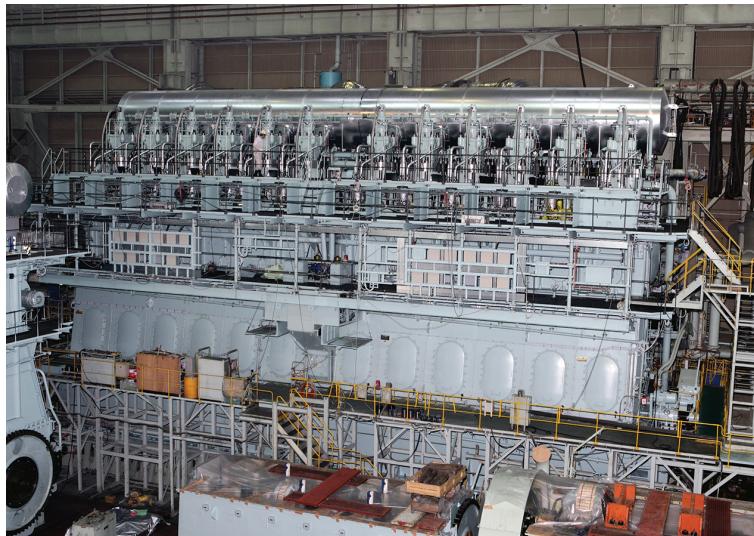


Most computers in the world are not immediately recognizable as computers. They are simply a part of a larger system or “gadget.” For example:

- *Cars*: A modern car may have many dozens of computers, controlling the fuel injection, monitoring engine performance, adjusting the radio, controlling the brakes, watching for underinflated tires, controlling the windshield wipers, etc.
- *Telephones*: A mobile telephone contains at least two computers; often one of those is specialized for signal processing.
- *Airplanes*: A modern airplane contains computers for everything from running the passenger entertainment system to wiggling the wing tips for optimal flight properties.
- *Cameras*: There are cameras with five processors and for which each lens even has its own separate processor.
- *Credit cards* (of the “smart card” variety)
- *Medical equipment monitors and controllers* (e.g., CAT scanners)
- *Elevators* (lifts)
- *PDAs* (Personal Digital Assistants)
- *Printer controllers*
- *Sound systems*
- *MP3 players*

- *Kitchen appliances* (such as rice cookers and bread machines)
- *Telephone switches* (typically consisting of thousands of specialized computers)
- *Pump controllers* (for water pumps and oil pumps, etc.)
- *Welding robots*: some for use in tight or dangerous places where a human welder cannot go
- *Wind turbines*: some capable of generating megawatts of power and 200m (650ft) tall
- *Sea-wall gate controllers*
- *Assembly-line quality monitors*
- *Bar code readers*
- *Car assembly robots*
- *Centrifuge controllers* (as used in many medical analysis processes)
- *Disk-drive controllers*

These computers are parts of larger systems. Such “large systems” usually don’t look like computers and we don’t usually think of them as computers. When we see a car coming down the street, we don’t say, “Look, there’s a distributed computer system!” Well, the car is *also* a distributed computer system, but its operation is so integrated with the mechanical, electronic, and electrical parts that we can’t really consider the computers in isolation. The constraints on their computations (in time and space) and the very definition of program correctness cannot be separated from the larger system. Often, an embedded computer controls a physical device, and the correct behavior of the computer is defined as the correct operation of the physical device. Consider a large marine diesel engine:



Note the engineer at the head of cylinder number 5. This is a big engine, the kind of engine that powers the largest ships. If an engine like this fails, you'll read about it on the front page of your morning newspaper. On this engine, a cylinder control system, consisting of three computers, sits on each cylinder head. Each cylinder control system is connected to the engine control system (another three computers) through two independent networks. The engine control system is then connected to the control room where the engineers can communicate with it through a specialized GUI system. The complete system can also be remotely monitored via radio (through satellites) from a shipping-line control center. For more examples, see Chapter 1.

So, from a programmer's point of view, what's special about the programs running in the computers that are parts of that engine? More generally, what are examples of concerns that become prominent for various kinds of embedded systems that we don't typically have to worry too much about for "ordinary programs"?

- Often, *reliability is critical*: Failure can be spectacular, expensive (as in "billions of dollars"), and potentially lethal (for the people on board a wreck or the animals in its environment).
- Often, *resources (memory, processor cycles, power) are limited*: That's not likely to be a problem on the engine computer, but think of smartphones, sensors, computers on board space probes, etc. In a world where dual-processor 2GHz laptops with 8GB of memory are common, a critical computer in an airplane or a space probe may have just 60MHz and 256KB, and a small gadget just sub-1MHz and a few hundred words of RAM. Computers made resilient to environmental hazards (vibration, bumps, unstable electricity supplies, heat, cold, humidity, workers stepping on them, etc.) are typically far slower than what powers a student's laptop.
- Often, *real-time response is essential*: If the fuel injector misses an injection cycle, bad things can happen to a very complex system generating 100,000HP; miss a few cycles – that is, fail to function correctly for a second or so – and strange things can start happening to propellers that can be up to 33ft (10m) in diameter and weigh up to 130 tons. You really don't want that to happen.
- Often, *a system must function uninterrupted for years*: Maybe the system is running in a communications satellite orbiting the earth, or maybe the system is just so cheap and exists in so many copies that any significant repair rate would ruin its maker (think of MP3 players, credit cards with embedded chips, and automobile fuel injectors). In the United States, the mandated reliability criterion for backbone telephone switches is 20 minutes of downtime in 20 years (don't even think of taking such a switch down each time you want to change its program).

- Often, *hands-on maintenance is infeasible or very rare*: You can take a large ship into a harbor to service the computers every second year or so when other parts of the ship require service and the necessary computer specialists are available in the right place at the right time. Unscheduled, hands-on maintenance is infeasible (no bugs are allowed while the ship is in a major storm in the middle of the Pacific). You simply can't send someone to repair a space probe in orbit around Mars.

Few systems suffer all of these constraints, and any system that suffers even one is the domain of experts. Our aim is not to make you an “instant expert”; attempting to do that would be quite silly and very irresponsible. Our aim is to acquaint you with the basic problems and the basic concepts involved in their solution so that you can appreciate some of the skills needed to build such systems. Maybe you could become interested in acquiring such valuable skills. People who design and implement embedded systems are critical to many aspects of our technological civilization. This is an area where a professional can do a lot of good.

Is this relevant to novices? To C++ programmers? Yes and yes. There are many more embedded systems processors than there are conventional PCs. A huge fraction of programming jobs relate to embedded systems programming, so your first real job may involve embedded systems programming. Furthermore, the list of examples of embedded systems that started this section is drawn from what I have personally seen done using C++.

## 25.2 Basic concepts

Much programming of computers that are part of an embedded system can be just like other programming, so most of the ideas presented in this book apply. However, the emphasis is often different: we must adjust our use of programming language facilities to the constraints of the task, and often we must manipulate our hardware at the lowest level:

- Correctness*: This is even more important than usual. “Correctness” is not just an abstract concept. In the context of an embedded system, what it means for a program to be correct becomes not just a question of producing the correct results, but also producing them at the right time, in the right order, and using only an acceptable set of resources. Ideally, the details of what constitutes correctness are carefully specified, but often such a specification can be completed only after some experimentation. Often, critical experiments can be performed only after the complete system (of which the computer running the program is a part) has been built. Completely specifying correctness for an embedded system can at the same time be extremely difficult and extremely important. Here, “extremely difficult” can mean “impossible given the time and resources available”;

we must try our best using all available tools and techniques. Fortunately, the range of specification, simulation, testing, and other techniques in a given area can be quite impressive. Here, “extremely important” can mean “failure leads to injury or ruin.”

- *Fault tolerance:* We must be careful to specify the set of conditions that a program is supposed to handle. For example, for an ordinary student program, you might find it unfair if we kicked the cord out of the power supply during a demonstration. Losing power is not among the conditions an ordinary PC application is supposed to deal with. However, losing power is not uncommon for embedded systems, and some are expected to deal with that. For example, a critical part of a system may have dual power sources, backup batteries, etc. Worse, “But I assumed that the hardware worked correctly” is no excuse for some applications. Over a long time and over a large range of conditions, hardware simply doesn’t work correctly. For example, some telephone switches and some aerospace applications are written based on the assumption that sooner or later some bit in the computer’s memory will just “decide” to change its value (e.g., from 0 to 1). Alternatively, it may “decide” that it likes the value 1 and ignore attempts to change that 1 to a 0. Such erroneous behavior happens eventually if you have enough memory and use it for a long enough time. It happens sooner if you expose the memory to hard radiation, such as you find beyond the earth’s atmosphere. When we work on a system (embedded or not), we have to decide what kind of tolerance to hardware failure we must provide. The usual default is to assume that hardware works as specified. As we deal with more critical systems, that assumption must be modified.
- *No downtime:* Embedded systems typically have to run for a long time without changes to the software or intervention by a skilled operator with knowledge of the implementation. “A long time” can be days, months, years, or the lifetime of the hardware. This is not unique for embedded systems, but it is a difference from the vast majority of “ordinary applications” and from all examples and exercises in this book (so far). This “must run forever” requirement implies an emphasis on error handling and resource management. What is a “resource”? A resource is something of which a machine has only a limited supply; from a program you acquire a resource through some explicit action (“acquire the resource,” “allocate”) and return it (“release,” “free,” “deallocate”) to the system explicitly or implicitly. Examples of resources are memory, file handles, network connections (sockets), and locks. A program that is part of a long-running system must release every resource it requires except a few that it permanently owns. For example, a program that forgets to close a

file every day will on most operating systems not survive for more than about a month. A program that fails to deallocate 100 bytes every day will waste more than 32K a year – that’s enough to crash a small gadget after a few months. The nasty thing about such resource “leaks” is that the program will work perfectly for months before it suddenly ceases to function. If a program will crash, we prefer it to crash as soon as possible so that we can remedy the problem. In particular, we prefer it to crash long before it is given to users.

- *Real-time constraints:* We can classify an embedded system as *hard real time* if a certain response must occur before a deadline. If a response must occur before a deadline most of the time, but we can afford an occasional time overrun, we classify the system as *soft real time*. Examples of soft real time are a controller for a car window and a stereo amplifier. A human will not notice a fraction of a second’s delay in the movement of the window, and only a trained listener would be able to hear a millisecond’s delay in a change of pitch. An example of hard real time is a fuel injector that has to “squirt” at exactly the right time relative to the movement of the piston. If the timing is off by even a fraction of a millisecond, performance suffers and the engine starts to deteriorate; a major timing problem could completely stop the engine, possibly leading to accident or disaster.
- *Predictability:* This is a key notion in embedded systems code. Obviously, the term has many intuitive meanings, but here – in the context of programming embedded systems – we will use a specialized technical meaning: an operation is *predictable* if it takes the same amount of time to execute every time it is executed on a given computer, and if all such operations take the same amount of time to execute. For example, when **x** and **y** are integers, **x+y** takes the same amount of time to execute every time and **xx+yy** takes the same amount of time when **xx** and **yy** are two other integers. Usually, we can ignore minor variations in execution speed related to machine architecture (e.g., differences caused by caching and pipelining) and simply rely on there being a fixed, constant upper limit on the time needed. Operations that are not predictable (in this sense of the word) can’t be used in hard real-time systems and must be used with great care in all real-time systems. A classic example of an unpredictable operation is a linear search of a list (e.g., **find()**) where the number of elements is unknown and not easily bounded. Only if we can reliably predict the number of elements or at least the maximum number of elements does such a search become acceptable in a hard real-time system; that is, to *guarantee* a response within a given fixed time we must be able to – possibly aided by code analysis tools – calculate the time needed for every possible code sequence leading up to the deadline.

- *Concurrency*: An embedded system typically has to respond to events from the external world. This leads to programs where many things happen “at once” because they correspond to real events that really happen at once. A program that simultaneously deals with several actions is called *concurrent* or *parallel*. Unfortunately the fascinating, difficult, and important issue of concurrency is beyond the scope of this book.

### 25.2.1 Predictability

From the point of view of predictability, C++ is pretty good, but it isn’t perfect. All facilities in the C++ language (including virtual function calls) are predictable, except

- Free-store allocation using `new` and `delete` (see §25.3)
- Exceptions (§19.5)
- `dynamic_cast` (§A.5.7)

These facilities must be avoided for hard real-time applications. The problems with `new` and `delete` are described in detail in §25.3; those are fundamental. Note that the standard library `string` and the standard containers (`vector`, `map`, etc.) indirectly use the free store, so they are not predictable either. The problem with `dynamic_cast` is a problem with current implementations but is not fundamental.

The problem with exceptions is that when looking at a particular `throw`, the programmer cannot – without looking at large sections of code – know how long it will take to find a matching `catch` or even if there is such a `catch`. In an embedded systems program, there had better be a `catch` because we can’t rely on a C++ programmer sitting ready to use the debugger. The problems with exceptions can in principle be dealt with by a tool that for each `throw` tells you exactly which `catch` will be invoked and how long it will take the `throw` to get there, but currently, that’s a research problem, so if you need predictability, you’ll have to make do with error handling based on return codes and other old-fashioned and tedious, but predictable, techniques.

### 25.2.2 Ideals

When writing an embedded systems program there is a danger that the quest for performance and reliability will lead the programmer to regress to exclusively using low-level language facilities. That strategy is workable for individual small pieces of code. However, it can easily leave the overall design a mess, make it difficult to be confident about correctness, and increase the time and money needed to build a system.

As ever, our ideal is to work at the highest level of abstraction that is feasible given the constraints on our problem. Don’t get reduced to writing glorified

assembler code! As ever, represent your ideas as directly in code as you can (given all constraints). As ever, try hard to write the clearest, cleanest, most maintainable code. Don't optimize until you have to. Performance (in time or space) is often essential for an embedded system, but trying to squeeze performance out of every little piece of code is misguided. Also, for many embedded systems the key is to be correct and fast enough; beyond "fast enough" the system simply idles until another action is needed. Trying to write every few lines of code to be as efficient as possible takes a lot of time, causes a lot of bugs, and often leads to missed opportunities for optimization as algorithms and data structures get hard to understand and hard to change. For example, that "low-level optimization" approach often leads to missed opportunities for memory optimization because almost similar code appears in many places and can't be shared because of incidental differences.

John Bentley – famous for his highly efficient code – offers two "laws of optimization":

- First law: Don't do it.
- Second law (for experts only): Don't do it yet.

Before optimizing, make sure that you understand the system. Only then can you be confident that it is – or can become – correct and reliable. Focus on algorithms and data structures. Once an early version of the system runs, carefully measure and tune it as needed. Fortunately, pleasant surprises are not uncommon: clean code sometimes runs fast enough and doesn't take up excessive memory space. Don't count on that, though; measure. Unpleasant surprises are not uncommon either.

### 25.2.3 Living with failure

Imagine that we are to design and implement a system that may not fail. By "not fail" let's say that we mean "will run without human intervention for a month." What kind of failures must we protect against? We can exclude dealing with the sun going nova and probably also with the system being trampled by an elephant. However, in general we cannot know what might go wrong. For a specific system, we can and must make assumptions about what kinds of errors are more common than others. Examples:

- Power surges/failure
- Connector vibrating out of its socket
- System hit by falling debris crushing a processor
- Falling system (disk might be destroyed by impact)
- X-rays causing some memory bits to change value in ways impossible according to the language definition

Transient errors are usually the hardest to find. A *transient error* is one that happens “sometimes” but not every time a program is run. For example, we have heard of a processor that misbehaved only when the temperature exceeded 130°F (54°C). It was never supposed to get that hot; however, it did when the system was (unintentionally and occasionally) covered up on the factory floor, never in the lab while being tested.

Errors that occur away from the lab are the hardest to fix. You will have a hard time imagining the design and implementation effort involved in letting the JPL engineers diagnose software and hardware failures on the Mars Rovers (20 minutes away from the lab for a signal traveling at the speed of light) and update the software to fix a problem once understood.

Domain knowledge – that is, knowledge about a system, its environment, and its use – is essential for designing and implementing a system with a good resilience against errors. Here, we will touch only upon generalities. Note that every “generality” we mention here has been the subject of thousands of papers and decades of research and development.

- *Prevent resource leaks:* Don’t leak. Be specific about what resources your program uses and be sure you conserve them (perfectly). Any leak will kill your system or subsystem eventually. The most fundamental resources are time and memory. Typically, a program will also use other resources, such as locks, communication channels, and files.
- *Replicate:* If a system critically needs a hardware resource (e.g., a computer, an output device, a wheel) to function, then the designer is faced with a basic choice: should the system contain several copies of the critical resource? We can either accept failure if the hardware breaks or provide a spare and let the software switch to using the spare. For example, the fuel injector controllers for the marine diesel engine are triplicate computers connected by duplicate networks. Note that “the spare” need not be identical to the original (e.g., a space probe may have a primary strong antenna and a weaker backup). Note also that “the spare” can typically be used to boost performance when the system works without a problem.
- *Self-check:* Know when the program (or hardware) is misbehaving. Hardware components (e.g., storage devices) can be very helpful in this respect, monitoring themselves for errors, correcting minor errors, and reporting major failures. Software can check for consistency of its data structures, check invariants (§9.4.3), and rely on internal “sanity checks” (assertions). Unfortunately, self-checking can itself be unreliable, and care must be taken that reporting an error doesn’t itself cause an error – it is really hard to completely check error checking.
- *Have a quick way out of misbehaving code:* Make systems modular. Base error handling on modules: each module has a specific task to do. If a module

decides it can't do its task, it can report that to some other module. Keep the error handling within a module simple (so that it is more likely to be correct and efficient), and have some other module responsible for serious errors. A good reliable system is modular and multi-level. At each level, serious errors are reported to a module at the next level – in the end, maybe to a person. A module that has been notified of a serious error (one that another module couldn't handle itself) can then take appropriate action – maybe involving a restart of the module that detected the error or running with a less sophisticated (but more robust) “backup” module. Defining exactly what “a module” is for a given system is part of the overall system design, but you can think of it as a class, a library, a program, or all the programs on a computer.

- *Monitor subsystems* in case they can't or don't notice a problem themselves. In a multi-level system higher levels can monitor lower levels. Many systems that really aren't allowed to fail (e.g., the marine engines or space station controllers) have three copies of critical subsystems. This triplication is not done just to have two spares, but also so that disagreements about which subsystem is misbehaving can be settled by 2-to-1 votes. Triplication is especially useful where a multi-level organization is difficult (i.e., at the highest level of a system or subsystem that may not fail).

We can design as much as we like and be as careful with the implementation as we know how to, but the system will still misbehave. Before delivering a system to users, it must be systematically and thoroughly tested; see Chapter 26.

## 25.3 Memory management

The two most fundamental resources in a computer are time (to execute instructions) and space (memory to hold data and code). In C++, there are three ways to allocate memory to hold data (§17.4, §A.4.2):

- *Static memory*: allocated by the linker and persisting as long as the program runs
- *Stack (automatic) memory*: allocated when we call a function and freed when we return from the function
- *Dynamic (heap) memory*: allocated by **new** and freed for possible reuse by **delete**

Let's consider these from the perspective of embedded systems programming. In particular, we will consider memory management from the perspective of tasks where predictability (§25.2.1) is considered essential, such as hard real-time programming and safety-critical programming.

Static memory poses no special problem in embedded systems programming: all is taken care of before the program starts to run and long before a system is deployed.

Stack memory can be a problem because it is possible to use too much of it, but this is not hard to take care of. The designers of a system must determine that for no execution of the program will the stack grow over an acceptable limit. This usually means that the maximum nesting of function calls must be limited; that is, we must be able to demonstrate that a chain of calls (e.g., **f1** calls **f2** calls ... calls **fn**) will never be too long. In some systems, that has caused a ban on recursive calls. Such a ban can be reasonable for some systems and for some recursive functions, but it is not fundamental. For example, I know that **factorial(10)** will call **factorial** at most ten times. However, an embedded systems programmer might very well prefer an iterative implementation of **factorial** (§15.5) to avoid any doubt or accident.

Dynamic memory allocation is usually banned or severely restricted; that is, **new** is either banned or its use restricted to a startup period, and **delete** is banned. The basic reasons are

- *Predictability:* Free-store allocation is not predictable; that is, it is not guaranteed to be a constant time operation. Usually, it is not: in many implementations of **new**, the time needed to allocate a new object can increase dramatically after many objects have been allocated and deallocated.
- *Fragmentation:* The free store may fragment; that is, after allocating and deallocating objects the remaining unused memory may be “fragmented” into a lot of little “holes” of unused space that are useless because each hole is too small to hold an object of the kind used by the application. Thus, the size of the useful free store can be far less than the size of the initial free store minus the size of the allocated objects.

The next section explains how this unacceptable state of affairs can arise. The bottom line is that we must avoid programming techniques that use both **new** and **delete** for hard real-time or safety-critical systems. The following sections explain how we can systematically avoid problems with the free store using stacks and pools.

### 25.3.1 Free-store problems

What’s the problem with **new**? Well, really it’s a problem with **new** and **delete** used together. Consider the result of this sequence of allocations and deallocations:

```
Message* get_input(Device&);           // make a Message on the free store

while(* . . .) {
    Message* p = get_input(dev);
    // . . .
```

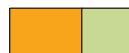
```

Node* n1 = new Node(arg1,arg2);
// ...
delete p;
Node* n2 = new Node (arg3,arg4);
// ...
}

```

Each time around the loop we create two **Nodes**, and in the process of doing so we create a **Message** and delete it again. Such code would not be unusual as part of building a data structure based on input from some “device.” Looking at this code, we might expect to “consume” **2\*sizeof(Node)** bytes of memory (plus free-store overhead) each time around the loop. Unfortunately, it is not guaranteed that the “consumption” of memory is restricted to the expected and desired **2\*sizeof(Node)** bytes. In fact, it is unlikely to be the case.

Assume a simple (though not unrealistic) memory manager. Assume also that a **Message** is a bit larger than a **Node**. We can visualize the use of free space like this, using orange for the **Message**, green for the **Nodes**, and plain white for “a hole” (that is, “unused space”):



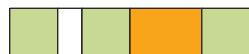
After creating **n1** (one **Message** and one **Node**)



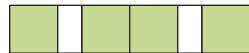
After deleting **p** (one “hole” and one **Node**)



After creating **n2** (two **Nodes** and a small “hole”)



After creating **n1** the 2nd time through the loop



After creating **n2** the 2nd time through the loop



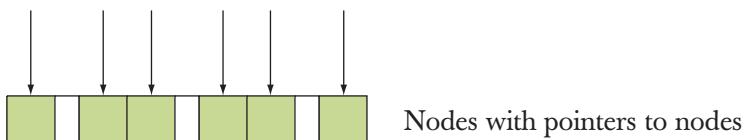
After creating **n2** the 3rd time through the loop

So, we are leaving behind some unused space (“a hole”) on the free store each time we execute the loop. That may be just a few bytes, but if we can’t use those holes it will be as bad as a memory leak – and even a small leak will eventually kill a long-running program. Having the free space in our memory scattered in many “holes” too small for allocating new objects is called *memory fragmentation*. Basically, the free-store manager will eventually use up all “holes” that are big enough to hold the kind of objects that the program uses, leaving only holes that are too small to be useful. This is a serious problem for essentially all

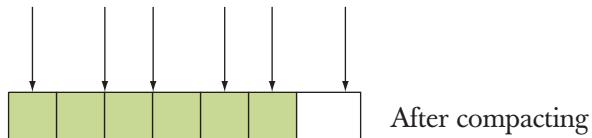
long-running programs that use `new` and `delete` extensively; it is not uncommon to find unusable fragments taking up most of the memory. That usually dramatically increases the time needed to execute `new` as it has to search through lots of objects and fragments for a suitably sized chunk of memory. Clearly this is not the kind of behavior we can accept for an embedded system. This can also be a serious problem in naively designed non-embedded systems.

Why can't "the language" or "the system" deal with this? Alternatively, can't we just write our program to not create such "holes"? Let's first examine the most obvious solution to having all those little useless "holes" in our memory: let's move the **Nodes** so that all the free space gets compacted into one contiguous area that we can use to allocate more objects.

Unfortunately, "the system" can't do that. The reason is that C++ code refers directly to objects in memory. For example, the pointers `n1` and `n2` contain real memory addresses. If we moved the objects pointed to, those addresses would no longer point to the right objects. Assume that we (somewhere) keep pointers to the nodes we created. We could represent the relevant part of our data structure like this:



Now we compact memory by moving an object so that all the unused memory is in one place:



Unfortunately, we now have made a mess of those pointers by moving the objects they pointed to without updating the pointers. Why don't we just update the pointers when we move the objects? We could write a program to do that, but only if we knew the details of the data structure. In general, "the system" (the C++ run-time support system) has no idea where the pointers are; that is, given an object, the question "Which pointers in the program point to this object right now?" has no good answer. Even if that problem could be easily solved, this approach (known as *compacting garbage collection*) is not always the right one. For example, to work well, it typically requires more than twice the memory that the program ever needs to be able to keep track of pointers and to move objects



around in. That extra memory may not be available on an embedded system. In addition, an efficient compacting garbage collector is hard to make predictable.

We could of course answer that “Where are the pointers?” question for our own data structures and compact those. That would work, but a simpler approach is to avoid fragmentation in the first place. In the example here, we could simply have allocated both **Nodes** before allocating the message:

```
while( . . . ) {  
    Node* n1 = new Node;  
    Node* n2 = new Node;  
    Message* p = get_input(dev);  
    // . . . store information in nodes . . .  
    delete p;  
    // . . .  
}
```

However, rearranging code to avoid fragmentation isn’t easy in general. Doing so reliably is at best very difficult and often incompatible with other rules for good code. Consequently, we prefer to restrict the use of the free store to ways that don’t cause fragmentation in the first place. Often, preventing a problem is better than solving it.

---

### TRY THIS



Complete the program above and print out the addresses and sizes of the objects created to see if and how “holes” appear on your machine. If you have time, you might draw memory layouts like the ones above to better visualize what’s going on.

#### 25.3.2 Alternatives to the general free store

So, we mustn’t cause fragmentation. What do we do then? The first simple observation is that **new** cannot by itself cause fragmentation; it needs **delete** to create the holes. So we start by banning **delete**. That implies that once an object is allocated, it will stay part of the program forever.

In the absence of **delete**, is **new** predictable; that is, do all **new** operations take the same amount of time? Yes, in all common implementations, but it is not actually guaranteed by the standard. Usually, an embedded system has a startup sequence of code that establishes the system as “ready to run” after initial power-up or restart. During that period, we can allocate memory any way we like



up to an allowed maximum. We could decide to use `new` during startup. Alternatively (or additionally) we could set aside global (static) memory for future use. For reasons of program structure, global data is often best avoided, but it can be sensible to use that language mechanism to pre-allocate memory. The exact rules for this should be laid down in a coding standard for a system (see §25.6).

There are two data structures that are particularly useful for predictable memory allocation:

- *Stacks*: A stack is a data structure where you can allocate an arbitrary amount of memory (up to a given maximum size) and deallocate the last allocation (only); that is, a stack can grow and shrink only at the top. There can be no fragmentation, because there can be no “hole” between two allocations.
- *Pools*: A pool is a collection of objects of the same size. We can allocate and deallocate objects as long as we don’t allocate more objects than the pool can hold. There can be no fragmentation because all objects are of the same size.

For both stacks and pools, both allocation and deallocation are predictable and fast.

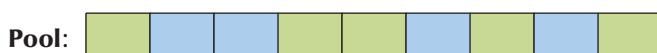
So, for a hard real-time or critical system we can define stacks and pools as needed. Better yet, we ought to be able to use stacks and pools as specified, implemented, and tested by someone else (as long as the specification meets our needs).

Note that the C++ standard containers (`vector`, `map`, etc.) and the standard `string` are not to be used because they indirectly use `new`. You can build (buy or borrow) “standard-like” containers to be predictable, but the default ones that come with your implementation are not constrained for embedded systems use.

Note that embedded systems typically have very stringent reliability requirements, so whatever solution we choose, we must make sure not to compromise our programming style by regressing into using lots of low-level facilities directly. Code that is full of pointers, explicit conversions, etc. is unreasonably hard to guarantee as correct.

### 25.3.3 Pool example

A *pool* is a data structure from which we can allocate objects of a given type and later deallocate (free) such objects. A pool contains a maximum number of objects; that number is specified when the pool is created. Using green for “allocated object” and blue for “space ready for allocation as an object,” we can visualize a pool like this:



A **Pool** can be defined like this:

```
template<typename T, int N>
class Pool {                                // Pool of N objects of type T
public:
    Pool();                                // make pool of N Ts
    T* get();                                // get a T from the pool; return 0 if no free Ts
    void free(T*);                            // return a T given out by get() to the pool
    int available() const;                  // number of free Ts
private:
    // space for T[N] and data to keep track of which Ts are allocated
    // and which are not (e.g., a list of free objects)
};
```

Each **Pool** object has a type of elements and a maximum number of objects. We can use a **Pool** like this:

```
Pool<Small_buffer,10> sb_pool;
Pool<Status_indicator,200> indicator_pool;

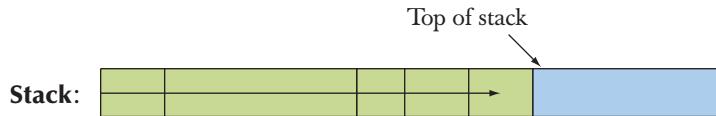
Small_buffer* p = sb_pool.get();
// ...
sb_pool.free(p);
```

It is the job of the programmer to make sure that a pool is never exhausted. The exact meaning of “make sure” depends on the application. For some systems, the programmer must write the code such that **get()** is never called unless there is an object to allocate. On other systems, a programmer can test the result of **get()** and take some remedial action if that result is **0**. A characteristic example of the latter is a telephone system engineered to handle at most 100,000 calls at a time. For each call, some resource, such as a dial buffer, is allocated. If the system runs out of dial buffers (e.g., **dial\_buffer\_pool.get()** returns **0**), the system refuses to set up new connections (and may “kill” a few existing calls to create capacity). The would-be caller can try again later.

Naturally, our **Pool** template is only one variation of the general idea of a pool. For example, where the restraints on memory allocation are less Draconian, we can define pools where the number of elements is specified in the constructor or even pools where the number of elements can be changed later if we need more objects than initially specified.

### 25.3.4 Stack example

A *stack* is a data structure from which we can allocate chunks of memory and deallocate the last allocated chunk. Using green for “allocated memory” and blue for “space ready for allocation,” we can visualize a stack like this:



As indicated, this stack “grows” toward the right.

We could define a stack of objects, just as we defined a pool of objects:

```
template<typename T, int N>
class Stack {                                // stack of N objects of type T
    // ...
};
```

However, most systems have a need for allocation of objects of varying sizes. A stack can do that whereas a pool cannot, so we’ll show how to define a stack from which we allocate “raw” memory of varying sizes rather than fixed-size objects:

```
template<int N>
class Stack {                                // stack of N bytes
public:
    Stack();                                // make an N-byte stack
    void* get(int n);                      // allocate n bytes from the stack;
                                                // return 0 if no free space
    void free();                            // return the last value returned by get() to the stack
    int available() const;                // number of available bytes
private:
    // space for char[N] and data to keep track of what is allocated
    // and what is not (e.g., a top-of-stack pointer)
};
```

Since **get()** returns a **void\*** pointing to the required number of bytes, it is our job to convert that memory to the kinds of objects we want. We can use such a stack like this:

```
Stack<50*1024> my_free_store; // 50K worth of storage to be used as a stack

void* pv1 = my_free_store.get(1024);
int* buffer = static_cast<int*>(pv1);
```

```
void* pv2 = my_free_store.get(sizeof(Connection));  
Connection* pconn = new(pv2) Connection(incoming,outgoing,buffer);
```

The use of `static_cast` is described in §17.8. The `new(pv2)` construct is a “placement `new`.” It means “Construct an object in the space pointed to by `pv2`.” It doesn’t allocate anything. The assumption here is that the type `Connection` has a constructor that will accept the argument list (`incoming,outgoing,buffer`). If that’s not the case, the program won’t compile.

Naturally, our `Stack` template is only one variation of the general idea of a stack. For example, where the restraints on memory allocation are less Draconian, we can define stacks where the number of bytes available for allocation is specified in the constructor.

## 25.4 Addresses, pointers, and arrays

Predictability is a need of some embedded systems; reliability is a concern of all. This leads to attempts to avoid language features and programming techniques that have proved error-prone (in the context of embedded systems programming, if not necessarily everywhere). Careless use of pointers is the main suspect here. Two problem areas stand out:

- Explicit (unchecked and unsafe) conversions
- Passing pointers to array elements

The former problem can typically be handled simply by severely restricting the use of explicit type conversions (casts). The pointer/array problems are more subtle, require understanding, and are best dealt with using (simple) classes or library facilities (such as `array`, §20.9). Consequently, this section focuses on how to address the latter problems.

### 25.4.1 Unchecked conversions

Physical resources (e.g., control registers for external devices) and their most basic software controls typically exist at specific addresses in a low-level system. We have to enter such addresses into our programs and give a type to such data. For example:

```
Device_driver* p = reinterpret_cast<Device_driver*>(0xffff8);
```

See also §17.8. This is the kind of programming you do with a manual or online documentation open. The correspondence between a hardware resource – the address of the resource’s register(s) (expressed as an integer, often a hexadecimal integer) – and pointers to the software that manipulates the hardware resource

is brittle. You have to get it right without much help from the compiler (because it is not a programming language issue). Usually, a simple (nasty, completely unchecked) `reinterpret_cast` from an `int` to a pointer type is the essential link in the chain of connections from an application to its nontrivial hardware resources.

Where explicit conversions (`reinterpret_cast`, `static_cast`, etc.; see §A.5.7) are not essential, avoid them. Such conversions (casts) are necessary far less frequently than is typically assumed by programmers whose primary experience is with C and C-style C++.

### 25.4.2 A problem: dysfunctional interfaces

As mentioned (§18.6.1), an array is often passed to a function as a pointer to an element (often, a pointer to the first element). Thereby, they “lose” their size, so that the receiving function cannot directly tell how many elements are pointed to, if any. This is a cause of many subtle and hard-to-fix bugs. Here, we examine examples of those array/pointer problems and present an alternative. We start with an example of a very poor (but unfortunately not rare) interface and proceed to improve it. Consider:

```
void poor(Shape* p, int sz)           // poor interface design
{
    for (int i = 0; i<sz; ++i) p[i].draw();
}

void f(Shape* q, vector<Circle>& s0) // very bad code
{
    Polygon s1[10];
    Shape s2[10];
    // initialize
    Shape* p1 = new Rectangle{Point{0,0},Point{10,20}};
    poor(&s0[0],s0.size());           // #1 (pass the array from the vector)
    poor(s1,10);                  // #2
    poor(s2,20);                  // #3
    poor(p1,1);                  // #4
    delete p1;
    p1 = 0;
    poor(p1,1);                  // #5
    poor(q,max);                 // #6
}
```

The function `poor()` is an example of poor interface design: it provides an interface that provides the caller ample opportunity for mistakes but offers the implementer essentially no opportunity to defend against such mistakes.

---

**TRY THIS**

Before reading further, try to see how many errors you can find in `f()`. Specifically, which of the calls of `poor()` could cause the program to crash?

---

At first glance, the calls look fine, but this is the kind of code that costs a programmer long nights of debugging and gives a quality engineer nightmares.

1. Passing the wrong element type, e.g., `poor(&s0[0],s0.size())`. Also, `s0` might be empty, in which case `&s0[0]` is wrong.
2. Use of a “magic constant” (here, correct): `poor(s1,10)`. Also, wrong element type.
3. Use of a “magic constant” (here, incorrect): `poor(s2,20)`.
4. Correct (easily verified): first call `poor(p1,1)`.
5. Passing a null pointer: second call `poor(p1,1)`.
6. May be correct: `poor(q,max)`. We can’t be sure from looking at this code fragment. To see if `q` points to an array with at least `max` elements, we have to find the definitions of `q` and `max` and determine their values at our point of use.

In each case, the errors are simple. We are not dealing with some subtle algorithmic or data structure problem. The problem is that `poor()`’s interface, involving an array passed as a pointer, opens the possibility of a collection of problems. You may appreciate how the problems were obscured by our use of “technical” unhelpful names, such as `p1` and `s0`. However, mnemonic, but misleading, names can make such problems even harder to spot.

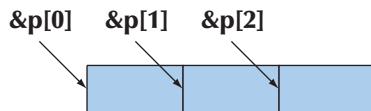
In theory, a compiler could catch a few of these errors (such as the second call of `poor(p1,1)` where `p1==0`), but realistically we are saved from disaster for this particular example only because the compiler catches the attempt to define objects of the abstract class `Shape`. However, that is unrelated to `poor()`’s interface problems, so we should not take too much comfort from that. In the following, we use a variant of `Shape` that is not abstract so as not to get distracted from the interface problems.

How come the `poor(&s0[0],s0.size())` call is an error? The `&s0[0]` refers to the first element of an array of `Circles`; it is a `Circle*`. We expect a `Shape*` and we pass a pointer to an object of a class derived from `Shape` (here, a `Circle*`). That’s obviously acceptable: we need that conversion so that we can do object-oriented programming, accessing objects of a variety of types through their common interface

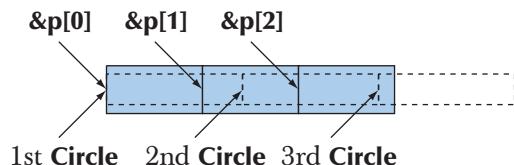
(here, **Shape**) (§14.2). However, **poor()** doesn't just use that **Shape\*** as a pointer; it uses it as an array, subscripting its way through that array:

```
for (int i = 0; i<sz; ++i) p[i].draw();
```

That is, it looks at the objects starting at memory locations **&p[0]**, **&p[1]**, **&p[2]**, etc.:



In terms of memory addresses, these pointers are **sizeof(Shape)** apart (§17.3.1). Unfortunately for **poor()**'s caller, **sizeof(Circle)** is larger than **sizeof(Shape)**, so that the memory layout can be visualized like this:



That is, **poor()** is calling **draw()** with a pointer into the middle of the **Circles**! This is likely to lead to immediate disaster (crash).

 The call **poor(s1,10)** is sneakier. It relies on a “magic constant” so it is immediately suspect as a maintenance hazard, but there is a deeper problem. The only reason the use of an array of **Polygons** doesn't immediately suffer the problem we saw for **Circles** is that a **Polygon** didn't add data members to its base class **Shape** (whereas **Circle** did; see §13.8 and §13.12); that is, **sizeof(Shape)==sizeof(Polygon)** and – more generally – a **Polygon** has the same memory layout as a **Shape**. In other words, we were “just lucky”; a slight change in the definition of **Polygon** will cause a crash. So **poor(s1,10)** works, but it is a bug waiting to happen. This is emphatically not quality code.

What we see here is the implementation reason for the general language rule that “a **D** is a **B**” does not imply “a **Container<D>** is a **Container<B>**” (§19.3.3). For example:

```
class Circle : public Shape {/* ... */};

void fv(vector<Shape>&);
void f(Shape &);
```

```
void g(vector<Circle>& vd, Circle & d)
{
    f(d);      // OK: implicit conversion from Circle to Shape
    fv(vd);   // error: no conversion from vector<Circle> to vector<Shape>
}
```

OK, so the use of **poor()** is very bad code, but can such code be considered embedded systems code; that is, should this kind of problem concern us in areas where safety or performance matters? Can we dismiss it as a hazard for programmers of non-critical systems and just tell them, “Don’t do that”? Well, many modern embedded systems rely critically on a GUI, which is almost always organized in the object-oriented manner of our example. Examples include the iPod user interface, the interfaces of some cell phones, and operator’s displays on “gadgets” up to and including airplanes. Another example is that controllers of similar gadgets (such as a variety of electric motors) can constitute a classic class hierarchy. In other words, this kind of code – and in particular, this kind of function declaration – is exactly the kind of code we should worry about. We need a safer way of passing information about collections of data without causing other significant problems.

So, we don’t want to pass a built-in array to a function as a pointer plus a size. What do we do instead? The simplest solution is to pass a reference to a container, such as a **vector**. The problems we saw for

```
void poor(Shape* p, int sz);
```

simply cannot occur for

```
void general(vector<Shape>&);
```

If you are programming where **std::vector** (or the equivalent) is acceptable, simply use **vector** (or the equivalent) consistently in interfaces; never pass a built-in array as a pointer plus a size.

If you can’t restrict yourself to **vector** or equivalents, you enter a territory that is more difficult and the solutions there involve techniques and language features that are not simple – even though the use of the class (**Array\_ref**) we provide is straightforward.

### 25.4.3 A solution: an interface class

Unfortunately, we cannot use **std::vector** in many embedded systems because it relies on the free store. We can solve that problem either by having a special implementation of **vector** or (more easily) by using a container that behaves like

a **vector** but doesn't do memory management. Before outlining such an interface class, let's consider what we want from it:

- It is a reference to objects in memory (it does not own objects, allocate objects, delete objects, etc.).
- It "knows" its size (so that it is potentially range checked).
- It "knows" the exact type of its elements (so that it cannot be the source of type errors).
- It is as cheap to pass (copy) as a (pointer, count) pair.
- It does *not* implicitly convert to a pointer.
- It is easy to express a subrange of the range of elements described by an interface object.
- It is as easy to use as built-in arrays.

We will only be able to approximate "as easy to use as built-in arrays." We don't want it to be so easy to use that errors start to become likely.

Here is one such class:

```
template<typename T>
class Array_ref {
public:
    Array_ref(T* pp, int s) :p{pp}, sz{s} {}

    T& operator[ ](int n) { return p[n]; }
    const T& operator[ ](int n) const { return p[n]; }

    bool assign(Array_ref a)
    {
        if (a.sz!=sz) return false;
        for (int i=0; i<sz; ++i) { p[i]=a.p[i]; }
        return true;
    }

    void reset(Array_ref a) { reset(a.p,a.sz); }
    void reset(T* pp, int s) { p=pp; sz=s; }

    int size() const { return sz; }

    // default copy operations:
    // Array_ref doesn't own any resources
    // Array_ref has reference semantics
```

```
private:
    T* p;
    int sz;
};
```

**Array\_ref** is close to minimal:

- No **push\_back()** (that would require the free store) and no **at()** (that would require exceptions).
- **Array\_ref** is a form of reference, so copying simply copies **(p,sz)**.
- By initializing with different arrays, we can have **Array\_refs** that are of the same type but have different sizes.
- By updating **(p,size)** using **reset()**, we can change the size of an existing **Array\_ref** (many algorithms require specification of subranges).
- No iterator interface (but that could be easily added if we needed it). In fact, an **Array\_ref** is in concept very close to a range described by two iterators.

An **Array\_ref** does not own its elements; it does no memory management; it is simply a mechanism for accessing and passing a sequence of elements. In that, it differs from the standard library **array** (§20.9).

To ease the creation of **Array\_refs**, we supply a few useful helper functions:

```
template<typename T> Array_ref<T> make_ref(T* pp, int s)
{
    return (pp) ? Array_ref<T>{pp,s} : Array_ref<T>{nullptr,0};
}
```

If we initialize an **Array\_ref** with a pointer, we have to explicitly supply a size. That's an obvious weakness because it provides us with an opportunity to give the wrong size. It also gives us an opportunity to use a pointer that is a result of an implicit conversion of an array of a derived class to a pointer to a base class, such as **Polygon[10]** to **Shape\*** (the original horrible problem from §25.4.2), but sometimes we simply have to trust the programmer.

We decided to be careful about null pointers (because they are a common source of problems), and we took a similar precaution for empty **vectors**:

```
template<typename T> Array_ref<T> make_ref(vector<T>& v)
{
    return (v.size()) ? Array_ref<T>{&v[0],v.size()} : Array_ref<T>{nullptr,0};
}
```

The idea is to pass the `vector`'s array of elements. We concern ourselves with `vector` here even though it is often not suitable in the kind of system where `Array_ref` can be useful. The reason is that it shares key properties with containers that can be used there (e.g., pool-based containers; see §25.3.3).

Finally, we deal with built-in arrays where the compiler knows the size:

```
template <typename T, int s> Array_ref<T> make_ref(T (&pp)[s])
{
    return Array_ref<T>{pp,s};
}
```

The curious `T(&pp)[s]` notation declares the argument `pp` to be a reference to an array of `s` elements of type `T`. That allows us to initialize an `Array_ref` with an array, remembering its size. We can't declare an empty array, so we don't have to test for zero elements:

```
Polygon ar[0];      // error: no elements
```

Given `Array_ref`, we can try to rewrite our example:

```
void better(Array_ref<Shape> a)
{
    for (int i = 0; i < a.size(); ++i) a[i].draw();
}

void f(Shape* q, vector<Circle>& s0)
{
    Polygon s1[10];
    Shape s2[20];
    // initialize
    Shape* p1 = new Rectangle{Point{0,0}, Point{10,20}};
    better(make_ref(s0));      // error: Array_ref<Shape> required
    better(make_ref(s1));      // error: Array_ref<Shape> required
    better(make_ref(s2));      // OK (no conversion required)
    better(make_ref(p1,1));    // OK: one element
    delete p1;
    p1 = 0;
    better(make_ref(p1,1));    // OK: no elements
    better(make_ref(q,max));  // OK (if max is OK)
}
```

We see improvements:

- The code is simpler. The programmer rarely has to think about sizes, but when necessary they are in a specific place (the creation of an `Array_ref`), rather than scattered throughout the code.
- The type problem with the `Circle[]`-to-`Shape[]` and `Polygon[]`-to-`Shape[]` conversions is caught.
- The problems with the wrong number of elements for `s1` and `s2` are implicitly dealt with.
- The potential problem with `max` (and other element counts for pointers) becomes more visible – it's the only place we have to be explicit about size.
- We deal implicitly and systematically with null pointers and empty `vectors`.

#### 25.4.4 Inheritance and containers

But what if we wanted to treat a collection of `Circles` as a collection of `Shapes`, that is, if we really wanted `better()` (which is a variant of our old friend `draw_all()`; see §19.3.2, §22.1.3) to handle polymorphism? Well, basically, we can't. In §19.3.3 and §25.4.2, we saw that the type system has very good reasons for refusing to accept a `vector<Circle>` as a `vector<Shape>`. For the same reason, it refuses to accept an `Array_ref<Circle>` as an `Array_ref<Shape>`. If you have a problem remembering why, it might be a good idea to reread §19.3.3, because the point is pretty fundamental even though it can be inconvenient.

Furthermore, to preserve run-time polymorphic behavior, we have to manipulate our polymorphic objects through pointers (or references): the dot in `a[i].draw()` in `better()` was a giveaway. We should have expected problems with polymorphism the second we saw that dot rather than an arrow (`->`). 

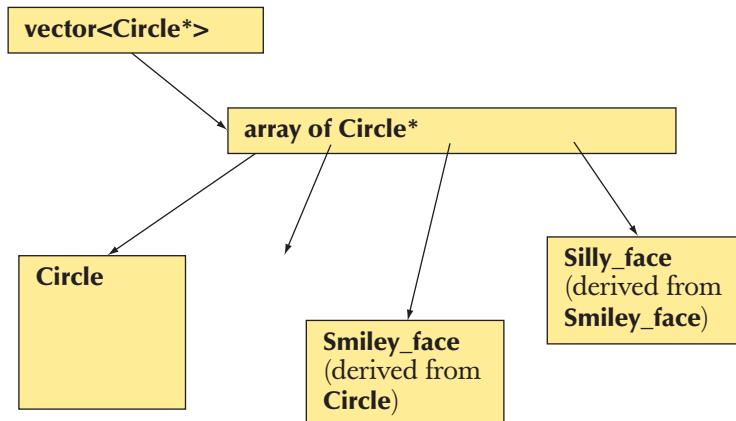
So what can we do? First we *must* use pointers (or references) rather than objects directly, so we'll try to use `Array_ref<Circle*>`, `Array_ref<Shape*>`, etc. rather than `Array_ref<Circle>`, `Array_ref<Shape>`, etc.

However, we still cannot convert an `Array_ref<Circle*>` to an `Array_ref<Shape*>` because we might then proceed to put elements into the `Array_ref<Shape*>` that are not `Circle*`s. But there is a loophole:

- Here, we don't want to modify our `Array_ref<Shape*>`; we just want to draw the `Shapes`! This is an interesting and useful special case: our argument against the `Array_ref<Circle*>`-to-`Array_ref<Shape*>` conversion doesn't apply to a case where we don't modify the `Array_ref<Shape*>`.
- All arrays of pointers have the same layout (independently of what kinds of objects they point to), so we don't get into the layout problem from §25.4.2.



That is, there would be nothing wrong with treating an `Array_ref<Circle*>` as an immutable `Array_ref<Shape*>`. So, we “just” have to find a way to treat an `Array_ref<Circle*>` as an immutable `Array_ref<Shape*>`. Consider:



There is no logical problem treating that array of `Circle*` as an immutable array of `Shape*` (from an `Array_ref`).



We seem to have strayed into expert territory. In fact, this problem is genuinely tricky and is unsolvable with the tools supplied so far. However, let’s see what it takes to produce a close-to-perfect alternative to our dysfunctional – but all too popular – interface style (pointer plus element count; see §25.4.2). Please remember: Don’t go into “expert territory” just to prove how clever you are. Most often, it is a better strategy to find a library where some experts have done the design, implementation, and testing for you.

First, we rework `better()` to something that uses pointers and guarantees that we don’t “mess with” the argument container:

```

void better2(const Array_ref<Shape*> const a)
{
    for (int i = 0; i < a.size(); ++i)
        if (a[i])
            a[i] -> draw();
}
  
```

We are now dealing with pointers, so we should check for null pointers. To make sure that `better2()` doesn’t modify our arrays and vectors in unsafe ways through `Array_ref`, we added a couple of `consts`. The first `const` ensures that we do not apply modifying (mutating) operations, such as `assign()` and `reset()`, on our

**Array\_ref**. The second **const** is placed after the \* to indicate that we want a constant pointer (rather than a pointer to constants); that is, we don't want to modify the element pointers even if we have operations available for that.

Next, we have to solve the central problem: how do we express the idea that **Array\_ref<Circle\*>** can be converted

- To something like **Array\_ref<Shape\*>** (that we can use in **better2()**)
- But only to an immutable version of **Array\_ref<Shape\*>**

We can do that by adding a conversion operator to **Array\_ref**:

```
template<typename T>
class Array_ref {
public:
    // as before

    template<typename Q>
    operator const Array_ref<const Q>()
    {
        // check implicit conversion of elements:
        static_cast<Q>(*static_cast<T*>(nullptr)); // check element
                                                       // conversion
        return Array_ref<const Q>{reinterpret_cast<Q*>(p),sz}; // convert
                                                               // Array_ref
    }

    // as before
};
```

This is headache-inducing, but basically:

- The operator casts to **Array\_ref<const Q>** for every type **Q** provided we can cast an element of **Array\_ref<T>** to an element of **Array\_ref<Q>** (we don't use the result of that cast; we just check that we can cast the element types).
- We construct a new **Array\_ref<const Q>** by using brute force (**reinterpret\_cast**) to get a pointer to the desired element type. Brute-force solutions often come at a cost; in this case, never use an **Array\_ref** conversion from a class using multiple inheritance (§A.12.4).
- Note that **const** in **Array\_ref<const Q>**: that's what ensures that we cannot copy an **Array\_ref<const Q>** into a plain old mutable **Array\_ref<Q>**.

We did warn you that this was “expert territory” and “headache-inducing.” However, this version of `Array_ref` is easy to use (it’s only the definition/implementation that is tricky):

```
void f(Shape* q, vector<Circle*>& s0)
{
    Polygon* s1[10];
    Shape* s2[20];
    // initialize
    Shape* p1 = new Rectangle{Point{0,0},10};
    better2(make_ref(s0));           // OK: converts to Array_ref<Shape*const>
    better2(make_ref(s1));           // OK: converts to Array_ref<Shape*const>
    better2(make_ref(s2));           // OK (no conversion needed)
    better2(make_ref(p1,1));          // error
    better2(make_ref(q,max));        // error
}
```

The attempts to use pointers result in errors because they are `Shape*`s whereas `better2()` expects an `Array_ref<Shape*>`; that is, `better2()` expects something that holds pointers rather than a pointer. If we want to pass pointers to `better2()`, we have to put them into a container (e.g., a built-in array or a `vector`) and pass that. For an individual pointer, we could use the awkward `make_ref(&p1,1)`. However, there is no solution for arrays (with more than one element) that doesn’t involve creating a container of pointers to objects.

In conclusion, we can create simple, safe, easy-to-use, and efficient interfaces to compensate for the weaknesses of arrays. That was the major aim of this section. “Every problem is solved by another indirection” (quote by David Wheeler) has been proposed as “the first law of computer science.” That was the way we solved this interface problem.

## 25.5 Bits, bytes, and words

We have talked about hardware memory concepts, such as bits, bytes, and words, before, but in general programming those are not the ones we think much about. Instead we think in terms of objects of specific types, such as `double`, `string`, `Matrix`, and `Simple_window`. Here, we will look at a level of programming where we have to be more aware of the realities of the underlying memory.

If you are uncertain about your knowledge of binary and hexadecimal representations of integers, this may be a good time to review §A.2.1.1.

### 25.5.1 Bits and bit operations

Think of a byte as a sequence of 8 bits:

7:	6:	5:	4:	3:	2:	1:	0:
1	0	1	0	0	1	1	1

Note the convention of numbering bits in a byte from the right (the least significant bit) to the left (the most significant bit). Now think of a word as a sequence of 4 bytes:

3:	2:	1:	0:
0xff	0x10	0xde	0xad

Again, we number right to left, that is, least significant byte to most significant byte. These pictures oversimplify what is found in the real world: there have been computers where a byte was 9 bits (but we haven't seen one for a decade), and machines where a word is 2 bytes are not rare. However, as long as you remember to check your system's manual before taking advantage of "8 bits" and "4 bytes," you should be fine.

In code meant to be portable, use [`limits`](#) (§24.2.1) to make sure your assumptions about sizes are correct. It is possible to place assertions in the code for the compiler to check:

```
static_assert(4<=sizeof(int),"ints are too small");
static_assert(!numeric_limits<char>::is_signed,"char is signed");
```

The first argument of a `static_assert` is a constant expression assumed to be true. If it is not true, that is, the assertion failed, the compiler writes the second argument, a string, as part of an error message.

How do we represent a set of bits in C++? The answer depends on how many bits we need and what kinds of operations we want to be convenient and efficient. We can use the integer types as sets of bits:

- `bool` – 1 bit, but takes up a whole byte of space
- `char` – 8 bits
- `short` – 16 bits
- `int` – typically 32 bits, but many embedded systems have 16-bit `ints`
- `long int` – 32 bits or 64 bits (but at least as many bits as `int`)
- `long long int` – 32 bits or 64 bits (but at least as many bits as `long`)

The sizes quoted are typical, but different implementations may have different sizes, so if you need to know, test. In addition, the standard library provides ways of dealing with bits:

- `std::vector<bool>` – when we need more than  $8 * \text{sizeof(long)}$  bits
- `std::bitset` – when we need more than  $8 * \text{sizeof(long)}$  bits
- `std::set` – an unordered collection of named bits (see §21.6.5)
- A file: lots of bits (see §25.5.6)

Furthermore, we can use two language features to represent bits:

- Enumerations (`enums`); see §9.5
- Bitfields; see §25.5.5

This variety of ways to represent “bits” reflects the fact that ultimately everything in computer memory is a set of bits, so people have felt the urge to provide a variety of ways of looking at bits, naming bits, and doing operations on bits. Note that the built-in facilities deal with a set of a fixed number of bits (e.g., 8, 16, 32, and 64) so that the computer can do logical operations on them at optimal speed using operations provided directly by hardware. In contrast, the standard library facilities provide an arbitrary number of bits. This may limit performance, but don’t prejudge efficiency issues: the library facilities can be – and often are – optimized to run well if you pick a number of bits that maps well to the underlying hardware.

Let’s first look at the integers. For these, C++ basically provides the bitwise logical operations that the hardware directly implements. These operations apply to each bit of their operands:

Bitwise operations	
	or
&	and
^	exclusive or
<<	left shift
>>	right shift
~	complement
	Bit <b>n</b> of <b>x y</b> is 1 if bit <b>n</b> of <b>x</b> or bit <b>n</b> of <b>y</b> is 1.
	Bit <b>n</b> of <b>x&amp;y</b> is 1 if bit <b>n</b> of <b>x</b> and bit <b>n</b> of <b>y</b> is 1.
	Bit <b>n</b> of <b>x^y</b> is 1 if bit <b>n</b> of <b>x</b> or bit <b>n</b> of <b>y</b> is 1 but not if both are 1.
	Bit <b>n</b> of <b>x&lt;&lt;s</b> is bit <b>n+s</b> of <b>x</b> .
	Bit <b>n</b> of <b>x&gt;&gt;s</b> is bit <b>n-s</b> of <b>x</b> .
	Bit <b>n</b> of <b>~x</b> is the opposite of bit <b>n</b> of <b>x</b> .

You might find the inclusion of “exclusive or” (^, sometimes called “xor”) as a fundamental operation odd. However, that’s the essential operation in much graphics and encryption code.

The compiler won't confuse a bitwise logical `<<` for an output operator, but you might. To avoid confusion, remember that an output operator takes an `ostream` as its left-hand operand, whereas a bitwise logical operator takes an integer as its left-hand operand.

Note that `&` differs from `&&` and `|` differs from `||` by operating individually on every bit of its operands (§A.5.5), producing a result with as many bits as its operands. In contrast, `&&` and `||` just return `true` or `false`.

Let's try a couple of examples. We usually express bit patterns using hexadecimal notation. For a half byte (4 bits) we have

Hex	Bits	Hex	Bits
<code>0x0</code>	<code>0000</code>	<code>0x8</code>	<code>1000</code>
<code>0x1</code>	<code>0001</code>	<code>0x9</code>	<code>1001</code>
<code>0x2</code>	<code>0010</code>	<code>0xa</code>	<code>1010</code>
<code>0x3</code>	<code>0011</code>	<code>0xb</code>	<code>1011</code>
<code>0x4</code>	<code>0100</code>	<code>0xc</code>	<code>1100</code>
<code>0x5</code>	<code>0101</code>	<code>0xd</code>	<code>1101</code>
<code>0x6</code>	<code>0110</code>	<code>0xe</code>	<code>1110</code>
<code>0x7</code>	<code>0111</code>	<code>0xf</code>	<code>1111</code>

For numbers up to 9 we could have used decimal, but using hexadecimal helps us to remember that we are thinking about bit patterns. For bytes and words, hexadecimal becomes really useful. The bits in a byte can be expressed as two hexadecimal digits. For example:

Hex byte	Bits
<code>0x00</code>	<code>0000 0000</code>
<code>0x0f</code>	<code>0000 1111</code>
<code>0xf0</code>	<code>1111 0000</code>
<code>0xff</code>	<code>1111 1111</code>
<code>0xaa</code>	<code>1010 1010</code>
<code>0x55</code>	<code>0101 0101</code>

So, using **unsigned** (§25.5.3) to keep things as simple as possible, we can write

```
unsigned char a = 0xaa;
unsigned char x0 = ~a;      // complement of a
```

a: 0xaa

$\sim a$ : 0x55

```
unsigned char b = 0x0f;
unsigned char x1 = a&b;    // a and b
```

a: 0xaa

b: 0xf

$a \& b$ : 0xa

```
unsigned char x2 = a^b;    // exclusive or: a xor b
```

a: 0xaa

b: 0xf

$a \wedge b$ : 0xa5

```
unsigned char x3 = a<<1;    // left shift 1
```

a: 0xaa

$a \ll 1$ : 0x54

Note that a 0 is “shifted in” from beyond bit 0 (the least significant bit) to fill up the byte. The leftmost bit (bit 7) simply disappears.

```
unsigned char x4 == a>>2;    // right shift 2
```

a: 0xaa

$a \gg 2$ : 0x2a

Note that two 0s are “shifted in” from beyond bit 7 (the most significant bit) to fill up the byte. The rightmost 2 bits (bit 1 and bit 0) simply disappear.

We can draw bit patterns like this and it is good to get a feel for bit patterns, but it soon becomes tedious. Here is a little program that converts integers to their bit representation:

```
int main()
{
    for (int i; cin>>i; )
        cout << dec << i << "==" 
        << hex << "0x" << i << "==" 
        << bitset<8*sizeof(int)>{i} << '\n';
}
```

To print the individual bits of the integer, we use a standard library **bitset**:

```
bitset<8*sizeof(int)>{i}
```

A **bitset** is a fixed number of bits. In this case, we use the number of bits in an **int** – **8\*sizeof(int)** – and initialize that **bitset** with our integer **i**.

### TRY THIS



Get the bits example to work and try out a few values to develop a feel for binary and hexadecimal representations. If you get confused about the representation of negative values, just try again after reading §25.5.3.

### 25.5.2 bitset

The standard library template class **bitset** from **<bitset>** is used to represent and manipulate sets of bits. Each **bitset** is of a fixed size, specified at construction:

```
bitset<4> flags;
bitset<128> dword_bits;
bitset<12345> lots;
```

A **bitset** is by default initialized to “all zeros” but is typically given an initializer; **bitset** initializers can be unsigned integers or strings of zeros and ones. For example:

```
bitset<4> flags = 0xb;
bitset<128> dword_bits {string{"10101010101010"}};
bitset<12345> lots;
```

Here **lots** will be all zeros, and **dword\_bits** will have 112 zeros followed by the 16 bits we explicitly specified. If you try to initialize with a string that has characters different from '**0**' and '**1**', a **std::invalid\_argument** exception is thrown:

```
string s;
cin>>s;
bitset<12345> my_bits{s}; // may throw std::invalid_argument
```

We can use the usual bit manipulation operators for **bitsets**. Assume that **b1**, **b2**, and **b3** are **bitsets**:

<b>b1 = b2&amp;b3;</b>	<i>// and</i>
<b>b1 = b2 b3;</b>	<i>// or</i>
<b>b1 = b2^b3;</b>	<i>// xor</i>
<b>b1 = ~b2;</b>	<i>// complement</i>
<b>b1 = b2&lt;&lt;2;</b>	<i>// shift left</i>
<b>b1 = b2&gt;&gt;3;</b>	<i>// shift right</i>

Basically, for bit operations (bitwise logical operations), a **bitset** acts like an **unsigned int** (§25.5.3) of an arbitrary, user-specified size. What you can do to an **unsigned int** (with the exception of arithmetic operations), you can do to a **bitset**. In particular, **bitsets** are useful for I/O:

```
cin>>b; // read a bitset from input
cout<<bitset<8>{'c'}; // output the bit pattern for the character 'c'
```

When reading into a **bitset**, an input stream looks for zeros and ones. Consider:

**10121**

This is read as **101**, leaving **21** unread in the stream.

As for a byte and a word, the bits of a **bitset** are numbered right to left (from the least significant bit toward the most significant), so that, for example, the numerical value of bit 7 is  $2^7$ :

7:	6:	5:	4:	3:	2:	1:	0:
<b>1</b>	<b>0</b>	<b>1</b>	<b>0</b>	<b>0</b>	<b>1</b>	<b>1</b>	<b>1</b>

For **bitsets**, the numbering is not just a convention because a **bitset** supports sub-scripting of bits. For example:

```

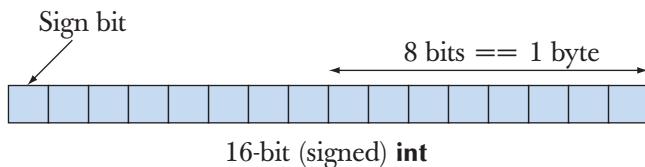
int main()
{
    constexpr int max = 10;
    for (bitset<max> b; cin>>b; ) {
        cout << b << '\n';
        for (int i =0; i<max; ++i) cout << b[i]; // reverse order
        cout << '\n';
    }
}

```

If you need a more complete picture of **bitsets**, look them up in your online documentation, a manual, or an expert-level textbook.

### 25.5.3 Signed and unsigned

Like most languages, C++ supports both signed and unsigned integers. Unsigned integers are trivial to represent in memory: bit0 means 1, bit1 means 2, bit2 means 4, and so on. However, signed integers pose a problem: how do we distinguish between positive and negative numbers? C++ gives the hardware designers some freedom of choice, but almost all implementations use the two's complement representation. The leftmost (most significant bit) is taken as the “sign bit”:



If the sign bit is 1, the number is negative. Almost universally, the two's complement representation is used. To save paper, we consider how we would represent signed numbers in a 4-bit integer:

<b>Positive:</b>	0	1	2	4	7
	<b>0000</b>	<b>0001</b>	<b>0010</b>	<b>0100</b>	<b>0111</b>
<b>Negative:</b>	<b>1111</b>	<b>1110</b>	<b>1101</b>	<b>1011</b>	<b>1000</b>
	-1	-2	-3	-5	-8

The bit pattern for **-(x+1)** can be described as the complement of the bits in **x** (also known as **~x**; see §25.5.1).

So far, we have just used signed integers (e.g., `int`). A slightly better set of rules would be:

- Use signed integers (e.g., `int`) for numbers.
- Use unsigned integers (e.g., `unsigned int`) for sets of bits.

That's not a bad rule of thumb, but it's hard to stick to because some people prefer unsigned integers for some forms of arithmetic and we sometimes need to use their code. In particular, for historical reasons going back to the early days of C when `ints` were 16 bits and every bit mattered, `v.size()` for a `vector` is an unsigned integer. For example:

```
vector<int> v;  
// ...  
for (int i = 0; i < v.size(); ++i) cout << v[i] << '\n';
```

A “helpful” compiler may warn us that we are mixing signed (i.e., `i`) and unsigned (i.e., `v.size()`) values. Mixing signed and unsigned variables could lead to disaster. For example, the loop variable `i` might overflow; that is, `v.size()` might be larger than the largest signed `int`. Then, `i` would reach the highest value that could represent a positive integer in a signed `int` (the number of bits in an `int` minus 1 to the power of two, minus 1, e.g.,  $2^{15}-1$ ). Then, the next `++` couldn't yield the next-highest integer and would instead result in a negative value. The loop would never terminate! Each time we reached the largest integer, we would start again from the smallest negative `int` value. So for 16-bit `ints` that loop is a (probably very serious) bug if `v.size()` is  $32*1024$  or larger; for 32-bit `ints` the problem occurs if `i` reaches  $2*1024*1024*1024$ .

So, technically, most of the loops in this book have been sloppy and could have caused problems. In other words, for an embedded system, we should either have verified that the loop could never reach the critical point or replaced it with a different form of loop. To avoid this problem we can use the `size_type` provided by `vector`, iterators, or a range-for-statement:

```
for (vector<int>::size_type i = 0; i < v.size(); ++i) cout << v[i] << '\n';  
  
for (vector<int>::iterator p = v.begin(); p != v.end(); ++p) cout << *p << '\n';  
  
for (int x : v) cout << x << '\n';
```

The `size_type` is guaranteed to be unsigned, so the first (unsigned integer) form has one more bit to play with than the `int` version above. That can be significant, but it still gives only a single bit of range (doubling the number of iterations that can be done). The loop using iterators has no such limitation.

---

**TRY THIS**

The following example may look innocent, but it is an infinite loop:

```
void infinite()
{
    unsigned char max = 160;      // very large
    for (signed char i=0; i<max; ++i) cout << int(i) << '\n';
}
```

Run it and explain why.

Basically, there are two reasons for using unsigned integers as integers, as opposed to using them simply as sets of bits (i.e., not using `+`, `-`, `*`, and `/`):

- To gain that extra bit of precision
- To express the logical property that the integer can't be negative

The former is what programmers get out of using an unsigned loop variable.

The problem with using both signed and unsigned types is that in C++ (as in C) they convert to each other in surprising and hard-to-remember ways. Consider:

`unsigned int ui = -1;`



```
int si = ui;
int si2 = ui+2;
unsigned ui2 = ui+2;
```



Surprisingly, the first initialization succeeds and `ui` gets the value 4294967295, which is the unsigned 32-bit integer with the same representation (bit pattern) as the signed integer `-1` (“all ones”). Some people consider that neat and use `-1` as shorthand for “all ones”; others consider that a problem. The same conversion rule applies from unsigned to signed, so `si` gets the value `-1`. As we would expect, `si2` becomes `1` ( $-1+2 == 1$ ), and so does `ui2`. The result for `ui2` ought to surprise you for a second: why should  $4294967295+2$  be `1`? Look at 4294967295 as a hexadecimal number (`0xffffffff`) and things become clearer: 4294967295 is the largest unsigned 32-bit integer, so 4294967297 cannot be represented as a 32-bit integer – unsigned or not. So we say either that 4294967295+2 overflowed or (more precisely) that unsigned integers support modular arithmetic; that is, arithmetic on 32-bit integers is modulo-32 arithmetic.

 Is everything clear so far? Even if it is, we hope we have convinced you that playing with that extra bit of precision in an unsigned integer is playing with fire. It can be confusing and is therefore a potential source of errors.

What happens if an integer overflows? Consider:

  
**Int i = 0;**  
**while (++i) print(i); // print i as an integer followed by a space**

What sequence of values will be printed? Obviously, this depends on the definition of **Int** (no, for once, the use of the capital *I* isn't a typo). For an integer type with a limited number of bits, we will eventually overflow. If **Int** is unsigned (e.g., **unsigned char**, **unsigned int**, or **unsigned long long**), the **++** is modulo arithmetic, so after the largest number that can be represented we get 0 (and the loop terminates). If **Int** is a signed integer (e.g., **signed char**), the numbers will suddenly turn negative and start working their way back up to 0 (where the loop will terminate). For example, for a **signed char**, we will see 1 2 . . . 126 127 -128 -127 . . . -2 -1.

What happens if an integer overflows? The answer is that we proceed as if we had enough bits, but throw away whichever part of the result doesn't fit in the integer into which we store our result. That strategy will lose us the leftmost (most significant) bits. That's the same effect we see when we assign:

```
int si = 257;           // doesn't fit into a char
char c = si;            // implicit conversion to char
unsigned char uc = si;
signed char sc = si;
print(si); print(c); print(uc); print(sc); cout << '\n';

si = 129;               // doesn't fit into a signed char
c = si;
uc = si;
sc = si;
print(si); print(c); print(uc); print(sc);
```

We get

257	1	1	1
129	-127	129	-127

The explanation of this result is that 257 is two more than will fit into 8 bits (255 is “8 ones”) and 129 is two more than can fit into 7 bits (127 is “7 ones”) so the sign bit gets set. Aside: This program shows that **chars** on our machine are signed (**c** behaves as **sc** and differs from **uc**).

---

**TRY THIS**

Draw out the bit patterns on a piece of paper. Using paper, then figure out what the answer would be for `si=128`. Then run the program to see if your machine agrees.

An aside: Why did we introduce that `print()` function? We could try

```
cout << i << ' ';
```

However, if `i` was a `char`, we would then output it as a character rather than an integer value. So, to treat all integer types uniformly, we defined

```
template<typename T> void print(T i) { cout << i << '\t'; }

void print(char i) { cout << int(i) << '\t'; }

void print(signed char i) { cout << int(i) << '\t'; }

void print(unsigned char i) { cout << int(i) << '\t'; }
```

To conclude: You can use unsigned integers exactly as signed integers (including ordinary arithmetic), but avoid that when you can because it is tricky and error-prone.

- Try never to use unsigned just to get another bit of precision.
- If you need one extra bit, you'll soon need another.

Unfortunately, you can't completely avoid unsigned arithmetic:

- Subscripting for standard library containers uses unsigned.
- Some people like unsigned arithmetic.

#### 25.5.4 Bit manipulation

Why do we actually manipulate bits? Well, most of us prefer not to. “Bit fiddling” is low-level and error-prone, so when we have alternatives, we take them. However, bits are both fundamental and very useful, so many of us can't just pretend they don't exist. This may sound a bit negative and discouraging, but that's deliberate. Some people really *love* to play with bits and bytes, so it is worth remembering that bit fiddling is something you do when you must (quite possibly having some fun in the process), but bits shouldn't be everywhere in your code.

To quote John Bentley: “People who play with bits will be bitten” and “People who play with bytes will be bytten.”

So, when do we manipulate bits? Sometimes the natural objects of our application simply are bits, so that some of the natural operations in our application domain are bit operations. Examples of such domains are hardware indicators (“flags”), low-level communications (where we have to extract values of various types out of byte streams), graphics (where we have to compose pictures out of several levels of images), and encryption (see the next section).

For example, consider how to extract (low-level) information from an integer (maybe because we wanted to transmit it as bytes, the way binary I/O does):

```
void f(short val)                                // assume 16-bit, 2-byte short integer
{
    unsigned char right = val&0xff;      // rightmost (least significant) byte
    unsigned char left = val>>8;        // leftmost (most significant) byte
    // ...
    bool negative = val&0x8000;       // sign bit
    // ...
}
```

Such operations are common. They are known as “shift and mask.” We “shift” (using `<<` or `>>`) to place the bits we want to consider to the rightmost (least significant) part of the word where they are easy to manipulate. We “mask” using and (`&`) together with a bit pattern (here `0xff`) to eliminate (set to zero) the bits we do not want in the result.

When we want to name bits, we often use enumerations. For example:

```
enum Printer_flags {
    acknowledge=1,
    paper_empty=1<<1,
    busy=1<<2,
    out_of_black=1<<3,
    out_of_color=1<<4,
    // ...
};
```

This defines each enumerator to have exactly the value that its name indicates:

<b>out_of_color</b>	<b>16</b>	<b>0x10</b>	<b>0001 0000</b>
<b>out_of_black</b>	<b>8</b>	<b>0x8</b>	<b>0000 1000</b>
<b>busy</b>	<b>4</b>	<b>0x4</b>	<b>0000 0100</b>
<b>paper_empty</b>	<b>2</b>	<b>0x2</b>	<b>0000 0010</b>
<b>acknowledge</b>	<b>1</b>	<b>0x1</b>	<b>0000 0001</b>

Such values are useful because they can be combined independently:

```
unsigned char x = out_of_color | out_of_black; // x becomes 24 (16+8)
x |= paper_empty; // x becomes 26 (24+2)
```

Note how `|=` can be read as “set a bit” (or as “set some bits”). Similarly, `&` can be read as “Is a bit set?” For example:

```
if (x & out_of_color) { // is out_of_color set? (yes, it is)
// ...
}
```

We can still use `&` to mask:

```
unsigned char y = x &(out_of_color | out_of_black); // y becomes 24
```

Now `y` has a copy of the bits from `x`'s positions 4 and 3 (`out_of_color` and `out_of_black`).

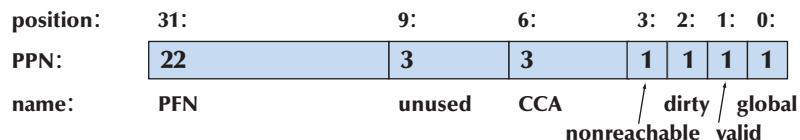
It is very common to use an `enum` as a set of bits. When doing that, we need a conversion to get the result of a bitwise logical operation “back into” the `enum`. For example:

```
Flags z = Printer_flags(out_of_color | out_of_black); // the cast is necessary
```

The reason that the cast is needed is that the compiler cannot know that the result of `out_of_color | out_of_black` is a valid value for a `Flags` variable. The compiler's skepticism is warranted: after all, no enumerator has a value 24 (`out_of_color | out_of_black`), but in this case, we know the assignment to be reasonable (but the compiler does not).

### 25.5.5 Bitfields

As mentioned, the hardware interface is one area where bits occur frequently. Typically, an interface is defined as a mixture of bits and numbers of various sizes. These “bits and numbers” are typically named and occur in specific positions of a word, often called a *device register*. C++ has a specific language facility to deal with such fixed layouts: *bitfields*. Consider a page number as used in the page manager deep in an operating system. Here is a diagram from an operating system manual:



The 32-bit word is used as two numeric fields (one of 22 bits and one of 3 bits) and four flags (1 bit each). The sizes and positions of these pieces of data are fixed. There is even an unused (and unnamed) “field” in the middle. We can express this as a **struct**:

```
struct PPN {                                // R6000 Physical Page Number
    unsigned int PFN : 22 ;                // Page Frame Number
    int : 3 ;                            // unused
    unsigned int CCA : 3 ;                // Cache Coherency Algorithm
    bool nonreachable : 1 ;
    bool dirty : 1 ;
    bool valid : 1 ;
    bool global : 1 ;
};
```

We had to read the manual to see that **PFN** and **CCA** should be interpreted as unsigned integers, but otherwise we could write out that **struct** directly from the diagram. Bitfields fill a word left to right. You give the number of bits as an integer value after a colon. You can't specify an absolute position (e.g., bit 8). If you “consume” more bits with bitfields than a word can hold, the fields that don't fit are put into the next word. Hopefully, that's what you want. Once defined, a bitfield is used exactly like other variables:

```
void part_of_VM_system(PPN * p )
{
    // ...
    if (p->dirty) { // contents changed
        // copy to disk
        p->dirty = 0 ;
    }
    // ...
}
```

Bitfields primarily save you the bother of shifting and masking to get to information placed in the middle of a word. For example, given a **PPN** called **pn** we could extract **CCA** like this:

```
unsigned int x = pn.CCA;           // extract CCA
```

Had we used an **int** called **pni** to represent the same bits, we could instead have written

```
unsigned int y = (pni>>4)&0x7; // extract CCA
```

That is, shift **pn** right so that **CCA** is the leftmost bit, then mask all other bits off with **0x7** (i.e., last three bits set). If you look at the machine code, you'll most likely find that the generated code is identical for those two lines.

The “acronym soup” (**CCA**, **PPN**, **PFN**) is typical of code at this level and makes little sense out of context.

### 25.5.6 An example: simple encryption

As an example of manipulation of data at the level of the data's representation as bits and bytes, let us consider a simple encryption algorithm: the Tiny Encryption Algorithm (TEA). It was originally written by David Wheeler of Cambridge University (§22.2.1). It is small but the protection against undesired decryption is excellent.

Don't look too hard at the code (unless you really want to and are willing to risk a headache). We present the code simply to give you the flavor of some real-world and useful bit manipulation code. If you want to make a study of encryption, you need a separate textbook for that. For more information and variants of the algorithm in other languages, see [http://en.wikipedia.org/wiki/Tiny\\_Encryption\\_Algorithm](http://en.wikipedia.org/wiki/Tiny_Encryption_Algorithm) and the TEA website of Professor Simon Shepherd, Bradford University, England. The code is not meant to be self-explanatory (no comments!).

The basic idea of enciphering/deciphering (also known as encryption/decryption) is simple. I want to send you some text, but I don't want others to read it. Therefore, I transform the text in a way that renders it unreadable to people who don't know exactly how I modified it – but in such a way that you can reverse my transformation and read the text. That's called enciphering. To encipher I use an algorithm (which we must assume an uninvited listener knows) and a string called the “key.” Both you and I have the key (and we hope that the uninvited listener does not). When you get the enciphered text, you decipher it using the “key”; that is, you reconstitute the “clear text” that I sent.

TEA takes as argument an array of two unsigned **longs** (**v[0],v[1]**) representing eight characters to be enciphered, an array of two unsigned **longs** (**w[0],w[1]**) into which the enciphered output is written, and an array of four unsigned **longs** (**k[0]..k[3]**), which is the key:

```
void encipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    static_assert(sizeof(long)==4,"size of long wrong for TEA");

    unsigned long y = v[0];
    unsigned long z = v[1];
```

```

unsigned long sum = 0;
const unsigned long delta = 0x9E3779B9;

for (unsigned long n = 32; n-->0; ) {
    y += (z<<4 ^ z>>5) + z^sum + k[sum&3];
    sum += delta;
    z += (y<<4 ^ y>>5) + y^sum + k[sum>>11 & 3];
}
w[0]=y;
w[1]=z;
}

```

Note how all data is unsigned so that we can perform bitwise operations on it without fear of surprises caused by special treatment related to negative numbers. Shifts (`<<` and `>>`), exclusive or (`^`), and bitwise and (`&`) do the essential work with an ordinary (unsigned) addition thrown in for good measure. This code is specifically written for a machine where there are 4 bytes in a `long`. The code is littered with “magic” constants (e.g., it assumes that `sizeof(long)` is 4). That’s generally not a good idea, but this particular piece of software fits on a single sheet of paper. As a mathematical formula, it fits on the back of an envelope or – as originally intended – in the head of a programmer with a good memory. David Wheeler wanted to be able to encipher things while he was traveling without bringing notes, a laptop, etc. In addition to being small, this code is also fast. The variable `n` determines the number of iterations: the higher the number of iterations, the stronger the encryption. To the best of our knowledge, for `n==32` TEA has never been broken.

Here is the corresponding deciphering function:

```

void decipher(
    const unsigned long *const v,
    unsigned long *const w,
    const unsigned long * const k)
{
    static_assert(sizeof(long)==4,"size of long wrong for TEA");

    unsigned long y = v[0];
    unsigned long z = v[1];
    unsigned long sum = 0xC6EF3720;
    const unsigned long delta = 0x9E3779B9;

```

```

// sum = delta<<5, in general sum = delta * n
for (unsigned long n = 32; n-- > 0; ) {
    z -= (y << 4 ^ y >> 5) + y ^ sum + k[sum>>11 & 3];
    sum -= delta;
    y -= (z << 4 ^ z >> 5) + z ^ sum + k[sum&3];
}
w[0]=y;
w[1]=z;
}

```

We can use TEA like this to produce a file to be sent over an unsafe connection:

```

int main()      // sender
{
    const int nchar = 2*sizeof(long);      // 64 bits
    const int kchar = 2*nchar;            // 128 bits

    string op;
    string key;
    string infile;
    string outfile;
    cout << "please enter input file name, output file name, and key:\n";
    cin >> infile >> outfile >> key;
    while (key.size()<kchar) key += '0'; // pad key
    ifstream inf(infile);
    ofstream outf(outfile);
    if (!inf || !outf) error("bad file name");

    const unsigned long* k =
        reinterpret_cast<const unsigned long*>(key.data());

    unsigned long outptr[2];
    char inbuf[nchar];
    unsigned long* inptr = reinterpret_cast<unsigned long*>(inbuf);
    int count = 0;

    while (inf.get(inbuf[count])) {
        outf << hex;                      // use hexadecimal output
        if (++count == nchar) {
            encipher(inptr,outptr,k);
            // pad with leading zeros:

```

```

        outf << setw(8) << setfill('0') << outptr[0] << ''
            << setw(8) << setfill('0') << outptr[1] << '';
        count = 0;
    }
}

if (count) { // pad
    while(count != nchar) inbuf[count++] = '0';
    encipher(inptr,outptr,k);
    outf << outptr[0] << '' << outptr[1] << '';
}
}

```

The essential piece of code is the **while**-loop; the rest is just support. The **while**-loop reads characters into the input buffer, **inbuf**, and every time it has eight characters as needed by TEA it passes them to **encipher()**. TEA doesn't care about characters; in fact, it has no idea what it is enciphering. For example, you could encipher a photo or a phone conversation. All TEA cares about is that it is given 64 bits (two unsigned **longs**) so that it can produce a corresponding 64 bits. So, we take a pointer to the **inbuf** and cast it to an unsigned **long\*** and pass that to TEA. We do the same for the key; TEA will use the first 128 bits (four unsigned **longs**) of the key, so we "pad" the user's input to be sure that there are 128 bits. The last statement pads the text with zeros to make up the multiple of 64 bits (8 bytes) required by TEA.

How do we transmit the enciphered text? We have a free choice, but since it is "just bits" rather than ASCII or Unicode characters, we can't really treat it as ordinary text. Binary I/O (see §11.3.2) would be an option, but here we decided to output the output words as hexadecimal numbers:

5b8fb57c	806fbcce	2db72335	23989d1d	991206bc	0363a308
8f8111ac	38f3f2f3	9110a4bb	c5e1389f	64d7efe8	ba133559
4cc00fa0	6f77e537	bde7925f	f87045f0	472bad6e	dd228bc3
a5686903	51cc9a61	fc19144e	d3bcde62	4fdb7dc8	43d565e5
f1d3f026	b2887412	97580690	d2ea4f8b	2d8fb3b7	936cfaf6d
6a13ef90	fd036721	b80035e1	7467d8d8	d32bb67e	29923fde
197d4cd6	76874951	418e8a43	e9644c2a	eb10e848	ba67dcdb8
7115211f	dbe32069	e4e92f87	8bf3e33e	b18f942c	c965b87a
44489114	18d4f2bc	256da1bf	c57b1788	9113c372	12662c23
eeb63c45	82499657	a8265f44	7c866aae	7c80a631	e91475e1
5991ab8b	6aedbb73	71b642c4	8d78f68b	d602bfe4	d1eadde7
55f20835	1a6d3a4b	202c36b8	66a1e0f2	771993f3	11d1d0ab

74a8cf4	4ce54f5a	e5fda09d	acbd110	259a1a19	b964a3a9
456fd8a3	1e78591b	07c8f5a2	101641ec	d0c9d7e1	60dbeb11
b9ad8e72	ad30b839	201fc553	a34a79c4	217ca84d	30f666c6
d018e61c	d1c94ea6	6ca73314	cd60def1	6e16870e	45b94dc0
d7b44fc4	96e0425a	72839f71	d5b6427c	214340f9	8745882f
0602c1a2	b437c759	ca0e3903	bd4d8460	edd0551e	31d34dd3
c3f943ed	d2cae477	4d9d0b61	f647c377	0d9d303a	ce1de974
f9449784	df460350	5d42b06c	d4dedb54	17811b5f	4f723692
14d67edb	11da5447	67bc059a	4600f047	63e439e3	2e9d15f7
4f21bbbe	3d7c5e9b	433564f5	c3ff2597	3a1ea1df	305e2713
9421d209	2b52384f	f78fbae7	d03c1f58	6832680a	207609f3
9f2c5a59	ee31f147	2ebc3651	e017d9d6	d6d60ce2	2be1f2f9
eb9de5a8	95657e30	cad37fda	7bce06f4	457daf44	eb257206
418c24a5	de687477	5c1b3155	f744fbff	26800820	92224e9d
43c03a51	d168f2d1	624c54fe	73c99473	1bce8fbb	62452495
5de382c1	1a789445	aa00178a	3e583446	dcbd64c5	dddale73
fa168da2	60bc109e	7102ce40	9fed3a0b	44245e5d	f612ed4c
b5c161f8	97ff2fc0	1dbf5674	45965600	b04c0afa	b537a770
9ab9bee7	1624516c	0d3e556b	6de6eda7	d159b10e	71d5c1a6
b8bb87de	316a0fc9	62c01a3d	0a24a51f	86365842	52dabf4d
372ac18b	9a5df281	35c9f8d7	07c8f9b4	36b6d9a5	a08ae934
239efba5	5fe3fa6f	659df805	faf4c378	4c2048d6	e8bf4939
31167a93	43d17818	998ba244	55dba8ee	799e07e7	43d26aef
d5682864	05e641dc	b5948ec8	03457e3f	80c934fe	cc5ad4f9
0dc16bb2	a50aa1ef	d62ef1cd	f8fbbf67	30c17f12	718f4d9a
43295fed	561de2a0				

---

### TRY THIS



The key was **bs**; what was the text?



Any security expert will tell you that it is a dumb idea to store clear text and enciphered files together and also express an opinion about padding, about using a two-letter key, etc., but this is a programming book, rather than a book on computer security.

We tested the programs by reading the enciphered text and getting the original back. When writing a program, it is always nice to be able to conduct a simple test of correctness.

Here is the central part of the deciphering program:

```
unsigned long inptr[2];
char outbuf[nchar+1];
outbuf[nchar]=0; // terminator
unsigned long* outptr = reinterpret_cast<unsigned long*>(outbuf);
inf.setf(ios_base::hex ,ios_base::basefield); // use hexadecimal input

while (inf>>inptr[0]>>inptr[1]) {
    decipher(inptr,outptr,k);
    outf<<outbuf;
}
```

Note the use of

```
inf.setf(ios_base::hex ,ios_base::basefield);
```

to read the hexadecimal numbers. For decryption, it's the output buffer, **outbuf**, that we treat as bits using a cast.

Is TEA an example of embedded systems programming? Not specifically, but you can imagine it being used wherever privacy is needed or financial transactions are conducted – that could include many “gadgets.” Anyway, TEA demonstrates many of the characteristics of good embedded systems code: it is based on a well-understood (mathematical) model that makes us confident about its correctness, it's small, it's fast, and it relies directly on hardware properties. The interface style of **encipher()** and **decipher()** is not quite to our taste. However, **encipher()** and **decipher()** were designed to be C as well as C++ functions, so no C++ facilities that are not also supported by C could be used. In addition, the many “magic constants” came from direct hand translation from the math.

## 25.6 Coding standards

There are many sources of errors. The most serious and hardest to remedy relate to high-level design decisions, such as overall error-handling strategies, conformance to certain standards (or lack thereof), algorithms, the representation of data, etc. These problems are *not* the ones we address here. Instead, we focus on errors that arise from code that is poorly written, that is, code that uses programming language facilities in unnecessarily error-prone ways or expresses ideas in ways that obscure their meaning.

Coding standards try to address the latter kinds of problems by defining a “house style” that guides programmers to a subset of the C++ language that is deemed appropriate for a given application. For example, a coding standard for

an embedded system involving hard real-time constraints or for a system needing to run “forever” may prohibit the use of **new**. Typically a coding standard also tries to ensure that code written by two programmers is more similar than if they had chosen freely from all possible styles. For example, a coding standard may require that **for**-statements be used for loops (thereby banning **while**-statements). This can make code more uniform, and in large projects that can be important for maintenance. Please note that a coding standard is aimed at improving code for a specific kind of programming given a specific kind of programmer. There is no one coding standard suitable for all C++ applications and all C++ programmers.

So, the problems that a coding standard tries to address are problems that arise from the way we express our solutions rather than the problems that arise from inherent complexities of the problem we are trying to solve with our application. We could say that coding standards are trying to address incidental complexities rather than inherent complexities.

The major sources of such incidental complexities are

- *Overly clever programmers*, who use features they don’t understand or delight in complicated solutions
- *Undereducated programmers*, who don’t use the most appropriate language and library features
- *Unnecessary variations in programming style*, causing code performing similar tasks to look different and confuse maintainers
- *Inappropriate programming language*, leading to use of language features that are poorly adapted to a particular application area or to a particular group of programmers
- *Insufficient library use*, leading to lots of ad hoc manipulation of low-level resources
- *Inappropriate coding standards*, causing extra work or prohibiting the best solution to some classes of problems, thus becoming a source of the kind of problems that the standards were introduced to solve

### 25.6.1 What should a coding standard be?

A good coding standard should help a programmer write good code; that is, it should help the programmer by giving answers to lots of little questions that each programmer would otherwise have to spend time deciding on a case-by-case basis. There is an old engineer’s proverb that says, “Form is liberating.” Ideally, a coding standard should be prescriptive, stating what should be done. That seems obvious, but many coding standards are simply a list of prohibitions, with no guidance about what to do after having obeyed a long list of *don’ts*. Just being told what not to do is rarely helpful and often annoying.

The rules of a good coding standard should be verifiable, preferably by a program; that is, once we have written the code, we should be able to look at it and easily answer the question, “Have I broken any rule of my coding standard?”

A good coding standard should present a rationale for the rules. Programmers should not just be told, “Because that’s the way we do it!” When they are, they resent it. Worse, programmers invariably try to subvert parts of a coding standard that they see as pointless and as preventing them from doing a good job. Don’t expect to like everything about a coding standard. Even the best coding standard is a compromise, and most prohibit certain practices assumed to cause problems – even if they never caused you a problem. For example, inconsistent naming rules are a source of confusion, but different people have strong attachments to some naming conventions and strong dislikes of others. For example, I consider the **CamelCodingStyle** of identifiers “pug ugly” and strongly prefer **underscore\_style** as cleaner and inherently more readable, and many people agree. On the other hand, many reasonable people disagree. Obviously, no naming standard can please everyone, but in this case, as in many others, a consistent style is definitely better than the lack of a standard.

To summarize:

- A good coding standard is designed for a specific application domain and a specific group of programmers.
- A good coding standard is prescriptive as well as restrictive.
  - Recommending some “foundation” library facilities is often the most effective use of prescriptive rules.
- A coding standard is a set of rules for what code should look like,
  - Typically specifying naming and indentation rules; e.g., “Use ‘Stroustrup layout.’”
  - Typically specifying a subset of a language; e.g., “Don’t use **new** or **throw**.”
  - Typically specifying rules for commenting; e.g., “Every function must have a comment explaining what it does.”
  - Often requiring the use of certain libraries; e.g., “Use **<iostream>** rather than **<stdio.h>**” or “Use **vector** and **string** rather than built-in arrays and C-style strings.”
- Common aims of most coding standards are to improve
  - Reliability
  - Portability
  - Maintainability
  - Testability

- Reusability
  - Extensibility
  - Readability
- A good coding standard is better than no standard. We wouldn't start a major (multi-person, multi-year) industrial project without one.
  - A poor coding standard can be worse than no standard. For example, C++ coding standards that restrict programming to something like the C subset do harm. Unfortunately, poor coding standards are not uncommon.
  - All coding standards are disliked by programmers, even the good ones. Most programmers want to write their code exactly the way they like it.



### 25.6.2 Sample rules

Here, we would like to give you a flavor of a coding standard by listing some rules. Naturally, we pick rules that we hope will be useful to you. However, we have never seen a real-world coding standard that could be described in fewer than 35 pages, and most are much longer. So, we don't try to give you a complete set of rules here. Furthermore, every good coding standard is designed for a particular application area and for a particular set of programmers. So, we don't make any pretenses of universality.

The rules are numbered and contain a (brief) rationale. Many rules contain examples for easier comprehension. We distinguish between *recommendations*, which a programmer may occasionally decide to ignore, and *firm rules*, which must be followed. In a real set of rules, a firm rule can usually be broken (only) with written permission from a supervisor. Each violation of a recommendation or a firm rule requires a comment in the code. Any exceptions to a rule can be listed in the rule. A firm rule is identified by a capital *R* in its number. A recommendation is identified by a lowercase *r* in its number.

The rules are classified as

- General
- Preprocessor
- Naming and layout
- Class rules
- Function and expression rules
- Hard real time
- Critical systems

The “hard real-time” and “critical systems” rules apply only to projects classified as such.

Compared to a good real-world coding standard, our terminology is under-specified (e.g., what does “critical” really mean?) and the rules overly terse. Similarities between these rules and the JSF++ rules (see §25.6.3) are not accidental; I helped formulate the JSF++ rules. However, the code examples in this book do not conform to the rules below – after all, the book code is not critical embedded systems code.

### *General rules*

**R100:** Any one function or class shall contain no more than 200 logical source lines of code (non-comments).

*Reason:* Long functions and long classes tend to be complex and therefore difficult to comprehend and test.

**r101:** Any one function or class should fit on a screen and serve a single logical purpose.

*Reason:* A programmer looking at only part of a function or class is more likely to overlook a problem. A function that tries to perform several logical functions is likely to be longer and more complex than one that doesn’t.

**R102:** All code shall conform to ISO/IEC 14882:2011(E) standard C++.

*Reason:* Language extensions or variations from ISO/IEC 14882 are likely to be less stable, to be less well specified, and to limit portability.

### *Preprocessor rules*

**R200:** No macros shall be used except for source control using **#ifdef** and **#ifndef**.

*Reason:* Macros don’t obey scope and type rules. Macro use is not obvious when visually examining source text.

**R201:** **#include** shall be used only to include header (\*.h) files.

*Reason:* **#include** is used to access interface declarations – not implementation details.

**R202:** All **#include** directives shall precede all non-preprocessor declarations.

*Reason:* An **#include** in the middle of a file is more likely to be overlooked by a reader and to cause inconsistencies from a name resolved differently in different places.

**R203:** Header files (\*.h) shall not contain non-**const** variable definitions or non-inline, non-template function definitions.

*Reason:* Header files should contain interface declarations – not implementation details. However, constants are often seen as part of the interface, some very simple functions need to be inline (and therefore in headers) for performance, and current template implementations require complete template definitions in headers.

### **Naming and layout**

**R300:** Indentations shall be used and be consistent within the same source file.

*Reason:* Readability and style.

**R301:** Each new statement starts on a new line.

*Reason:* Readability.

*Example:*

```
int a = 7; x = a+7; f(x,9); // violation
int a = 7;           // OK
x = a+7;           // OK
f(x,9);           // OK
```

*Example:*

```
if (p<q) cout << *p;      // violation
```

*Example:*

```
if (p<q)
    cout << *p; // OK
```

**R302:** Identifiers should be given descriptive names.

Identifiers may contain common abbreviations and acronyms.

When used conventionally, **x**, **y**, **i**, **j**, etc. are descriptive.

Use the **number\_of\_elements** style rather than the **numberOfElements** style.

Hungarian notation shall not be used.

Type, template, and namespace names (only) start with a capital letter.

Avoid excessively long names.

*Example:* **Device\_driver** and **Buffer\_pool**.

*Reason:* Readability.

*Note:* Identifiers starting with an underscore are reserved to the language implementation by the C++ standard and thus banned.

*Exception:* When calling an approved library, the names from that library may be used.

**R303:** Identifiers shall not differ only by

- A mixture of case
- The presence/absence of the underscore character
- The interchange of the letter *O* with the number 0 or the letter *D*
- The interchange of the letter *I* with the number 1 or the letter *l*
- The interchange of the letter *S* with the number 5
- The interchange of the letter *Z* with the number 2
- The interchange of the letter *n* with the letter *h*

*Example:* **Head** and **head** // violation

*Reason:* Readability.

**R304:** No identifier shall be in all capital letters and underscores.

*Example:* **BLUE** and **BLUE\_CHEESE** // violation

*Reason:* All capital letters are widely used for macros that may be used in **#include** files for approved libraries.

*Exception:* Macro names used for **#include** guards.

#### *Function and expression rules*

**r400:** Identifiers in an inner scope should not be identical to identifiers in an outer scope.

*Example:*

**int var = 9;** { **int var = 7;** **++var;** } // violation: var hides var

*Reason:* Readability.

**R401:** Declarations shall be declared in the smallest possible scope.

*Reason:* Keeping initialization and use close minimizes chances of confusion; letting a variable go out of scope releases its resources.

**R402:** Variables shall be initialized.

*Example:*

**int var;** // violation: var is not initialized

*Reason:* Uninitialized variables are a common source of errors.

*Exception:* A variable that is immediately filled from input need not be initialized.

*Note:* Many types, such as **vector** and **string**, have a default constructor to guarantee initialization.

**R403:** Casts shall not be used.

*Reason:* Casts are a common source of errors.

*Exception:* `dynamic_cast` may be used.

*Exception:* Named casts may be used to convert hardware addresses into pointers and `void*` received from sources external to a program (e.g., a GUI library) into pointers of a proper type.

**R404:** Built-in arrays shall not be used in interfaces; that is, a pointer as function argument shall be assumed to point to a single element. Use `Array_ref` to pass arrays.

*Reason:* An array is passed as a pointer and its number of elements is not carried along to the called function. Also, the combination of implicit array-to-pointer conversion and implicit derived-to-base conversion can lead to memory corruption.

#### *Class rules*

**R500:** Use `class` for classes with no public data members. Use `struct` for classes with no private data members. Don't use classes with both public and private data members.

*Reason:* Clarity.

**r501:** If a class has a destructor or a member of pointer or reference type, it must have a copy constructor and a copy assignment defined or prohibited.

*Reason:* A destructor usually releases a resource. The default copy semantics rarely does "the right thing" for pointer and reference members or for a class with a destructor.

**R502:** If a class has a virtual function it must have a virtual destructor.

*Reason:* A class has a virtual function so that it can be used through a base class interface. A function that knows an object only through that base class may delete it and derived classes need a chance to clean up (in their destructors).

**r503:** A constructor that accepts a single argument must be declared `explicit`.

*Reason:* To avoid surprising implicit conversions.

#### *Hard real-time rules*

**R800:** Exceptions shall not be used.

*Reason:* Not predictable.

**R801:** `new` shall be used only during startup.

*Reason:* Not predictable.

*Exception:* Placement-`new` (with the standard meaning) may be used for memory allocated from stacks.

**R802:** `delete` shall not be used.

*Reason:* Not predictable; can cause fragmentation.

**R803:** `dynamic_cast` shall not be used.

*Reason:* Not predictable (assuming common implementation technique).

**R804:** The standard library containers, except `std::array`, shall not be used.

*Reason:* Not predictable (assuming common implementation technique).

### *Critical systems rules*

**R900:** Increment and decrement operations shall not be used as sub-expressions.

*Example:*

```
int x = v[++i]; // violation
```

*Example:*

```
++i;  
int x = v[i]; // OK
```

*Reason:* Such an increment might be overlooked.

**R901:** Code should not depend on precedence rules below the level of arithmetic expressions.

*Example:*

```
x = a*b+c; // OK
```

*Example:*

```
if ( a<b || c<=d ) // violation: parenthesize(a<b) and (c<=d)
```

*Reason:* Confusion about precedence has been repeatedly found in code written by programmers with a weak C/C++ background.

We left gaps in the numbering so that we could add new rules without changing the numbering of existing ones and still have the general classification recognized

through the numbering. It is very common for rules to become known by their number, so that renumbering would be resisted by the users.

### 25.6.3 Real coding standards

There are lots of C++ coding standards. Most are corporate and not widely available. In many cases, that's probably a good thing except possibly for the programmers of those corporations. Here is a list of standards that – when used appropriately in areas to which they apply – can do some good:

Google C++ Style Guide: <http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml>. A rather old-style and restrictive but evolving style guide.

Lockheed Martin Corporation. *Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program*. Document Number 2RDU00001 Rev C. December 2005. Colloquially known as “JSF++”; a set of rules written at Lockheed-Martin Aero for air vehicle (read “airplane”) software. These rules really were written by and for programmers who produce software upon which human lives depend. [www.stroustrup.com/JSF-AV-rules.pdf](http://www.stroustrup.com/JSF-AV-rules.pdf).

Programming Research. High-integrity C++ Coding Standard Manual Version 2.4. [www.programmingresearch.com](http://www.programmingresearch.com).

Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586. This is more of a “meta coding standard”; that is, instead of specific rules it has guidance on which rules are good and why.

Note that there is no substitute for knowing your application area, your programming language, and the relevant programming technique. For most applications – and certainly for most embedded systems programming – you also need to know your operating system and/or hardware architecture. If you need to use C++ for low-level coding, have a look at the ISO C++ committee’s report on performance (ISO/IEC TR 18015, [www.stroustrup.com/performanceTR.pdf](http://www.stroustrup.com/performanceTR.pdf)); by “performance” they/we primarily mean “embedded systems programming.”

Language dialects and proprietary languages abound in the embedded systems world, but whenever you can, use standardized language (such as ISO C++), tools, and libraries. That will minimize your learning curve and increase the likelihood that your work will last.



## Drill

1. Run this:

```
int v = 1; for (int i = 0; i<sizeof(v)*8; ++i) { cout << v << ' '; v <<=1; }
```

2. Run that again with `v` declared to be an `unsigned int`.
3. Using hexadecimal literals, define `short unsigned ints` with:
  - a. Every bit set
  - b. The lowest (least significant bit) set
  - c. The highest (most significant bit) set
  - d. The lowest byte set
  - e. The highest byte set
  - f. Every second bit set (and the lowest bit 1)
  - g. Every second bit set (and the lowest bit 0)
4. Print each as a decimal and as a hexadecimal.
5. Do 3 and 4 using bit manipulation operations (`|`, `&`, `<<`) and (only) the literals 1 and 0.

## Review

1. What is an embedded system? Give ten examples, out of which at least three should not be among those mentioned in this chapter.
2. What is special about embedded systems? Give five concerns that are common.
3. Define predictability in the context of embedded systems.
4. Why can it be hard to maintain and repair an embedded system?
5. Why can it be a poor idea to optimize a system for performance?
6. Why do we prefer higher levels of abstraction to low-level code?
7. What are transient errors? Why do we particularly fear them?
8. How can we design a system to survive failure?
9. Why can't we prevent every failure?
10. What is domain knowledge? Give examples of application domains.
11. Why do we need domain knowledge to program embedded systems?
12. What is a subsystem? Give examples.
13. From a C++ language point of view, what are the three kinds of storage?
14. When would you like to use the free store?
15. Why is it often infeasible to use the free store in an embedded system?
16. When can you safely use `new` in an embedded system?
17. What is the potential problem with `std::vector` in the context of embedded systems?

18. What is the potential problem with exceptions in the context of embedded systems?
19. What is a recursive function call? Why do some embedded systems programmers avoid them? What do they use instead?
20. What is memory fragmentation?
21. What is a garbage collector (in the context of programming)?
22. What is a memory leak? Why can it be a problem?
23. What is a resource? Give examples.
24. What is a resource leak and how can we systematically prevent it?
25. Why can't we easily move objects from one place in memory to another?
26. What is a stack?
27. What is a pool?
28. Why doesn't the use of stacks and pools lead to memory fragmentation?
29. Why is `reinterpret_cast` necessary? Why is it nasty?
30. Why are pointers dangerous as function arguments? Give examples.
31. What problems can arise from using pointers and arrays? Give examples.
32. What are alternatives to using pointers (to arrays) in interfaces?
33. What is "the first law of computer science"?
34. What is a bit?
35. What is a byte?
36. What is the usual number of bits in a byte?
37. What operations do we have on sets of bits?
38. What is an "exclusive or" and why is it useful?
39. How can we represent a set (sequence, whatever) of bits?
40. How do we conventionally number bits in a word?
41. How do we conventionally number bytes in a word?
42. What is a word?
43. What is the usual number of bits in a word?
44. What is the decimal value of `0xf7`?
45. What sequence of bits is `0xab`?
46. What is a `bitset` and when would you need one?
47. How does an `unsigned int` differ from a `signed int`?
48. When would you prefer an `unsigned int` to a `signed int`?
49. How would you write a loop if the number of elements to be looped over was very high?
50. What is the value of an `unsigned int` after you assign `-3` to it?
51. Why would we want to manipulate bits and bytes (rather than higher-level types)?
52. What is a bitfield?
53. For what are bitfields used?
54. What is encryption (enciphering)? Why do we use it?
55. Can you encrypt a photo?

56. What does TEA stand for?
57. How do you write a number to output in hexadecimal notation?
58. What is the purpose of coding standards? List reasons for having them.
59. Why can't we have a universal coding standard?
60. List some properties of a good coding standard.
61. How can a coding standard do harm?
62. Make a list of at least ten coding rules that you like (have found useful).  
    Why are they useful?
63. Why do we avoid ALL\_CAPITAL identifiers?

## Terms

address	encryption	pool
bit	exclusive or	predictability
bitfield	gadget	real time
<b>bitset</b>	garbage collector	resource
coding standard	hard real time	soft real time
embedded system	leak	<b>unsigned</b>

## Exercises

1. If you haven't already, do the **Try this** exercises in this chapter.
2. Make a list of words that can be spelled with hexadecimal notation. Read 0 as *o*, read 1 as *l*, read 2 as *to*, etc.; for example, Foo1 and Beef. Kindly eliminate vulgarities from the list before submitting it for grading.
3. Initialize a 32-bit signed integer with the bit patterns and print the result: all zeros, all ones, alternating ones and zeros (starting with a leftmost one), alternating zeros and ones (starting with a leftmost zero), the 110011001100 . . . pattern, the 001100110011 . . . pattern, the pattern of all-one bytes and all-zero bytes starting with an all-one byte, the pattern of all-one bytes and all-zero bytes starting with an all-zero byte. Repeat that exercise with a 32-bit unsigned integer.
4. Add the bitwise logical operators `&`, `|`, `^`, and `~` to the calculator from Chapter 7.
5. Write an infinite loop. Execute it.
6. Write an infinite loop that is hard to recognize as an infinite loop. A loop that isn't really infinite because it terminates after completely consuming some resource is acceptable.
7. Write out the hexadecimal values from 0 to 400; write out the hexadecimal values from -200 to 200.
8. Write out the numerical values of each character on your keyboard.
9. Without using any standard headers (such as `<limits>`) or documentation, compute the number of bits in an `int` and determine whether `char` is signed or unsigned on your implementation.

10. Look at the bitfield example from §25.5.5. Write an example that initializes a **PPN**, then reads and prints each field value, then changes each field value (by assigning to the field) and prints the result. Repeat this exercise, but store the **PPN** information in a 32-bit unsigned integer and use bit manipulation operators (§25.5.4) to access the bits in the word.
11. Repeat the previous exercise, but keep the bits in a **bitset<32>**.
12. Write out the clear text of the example from §25.5.6.
13. Use TEA (§25.5.6) to communicate “securely” between two computers. Email is minimally acceptable.
14. Implement a simple **vector** that can hold at most  $N$  elements allocated from a pool. Test it for  $N=1000$  and integer elements.
15. Measure the time (§26.6.1) it takes to allocate 10,000 objects of random sizes in the [1000:0)-byte range using **new**; then measure the time it takes to deallocate them using **delete**. Do this twice, once deallocating in the reverse order of allocation and once deallocating in random order. Then, do the equivalent for allocating 10,000 objects of size 500 bytes from a pool and freeing them. Then, do the equivalent of allocating 10,000 objects of random sizes in the [1000:0)-byte range on a stack and then free them (in reverse order). Compare the measurements. Do each measurement at least three times to make sure the results are consistent.
16. Formulate 20 coding style rules (don’t just copy those in §25.6). Apply them to a program of more than 300 lines that you recently wrote. Write a short (a page or two) comment on the experience of applying those rules. Did you find errors in the code? Did the code get clearer? Did some code get less clear? Now modify the set of rules based on this experience.
17. In §25.4.3–4 we provided a class **Array\_ref** claimed to make access to elements of an array simpler and safer. In particular, we claimed to handle inheritance correctly. Try a variety of ways to get a **Rectangle\*** into a **vector<Circle\*>** using an **Array\_ref<Shape\*>** but no casts or other operations involving undefined behavior. This ought to be impossible.

## Postscript

So, is embedded systems programming basically “bit fiddling”? Not at all, especially if you deliberately try to minimize bit fiddling as a potential problem with correctness. However, somewhere in a system bits and bytes have “to be fiddled”; the question is just where and how. In most systems, the low-level code can and should be localized. Many of the most interesting systems we deal with are embedded, and some of the most interesting and challenging programming tasks are in this field.







# Testing

“I have only proven the code correct, not tested it.”

—Donald Knuth

This chapter covers testing and design for correctness. These are huge topics, so we can only scratch their surfaces. The emphasis is on giving some practical ideas and techniques for testing units, such as functions and classes, of a program. We discuss the use of interfaces and the selection of tests to run against them. We emphasize the importance of designing systems to simplify testing and the use of testing from the earliest stages of development. Proving programs correct and dealing with performance problems are also briefly considered.

<b>26.1 What we want</b>	<b>26.4 Design for testing</b>
26.1.1 Caveat	26.5 Debugging
<b>26.2 Proofs</b>	<b>26.6 Performance</b>
<b>26.3 Testing</b>	26.6.1 Timing
26.3.1 Regression tests	<b>26.7 References</b>
26.3.2 Unit tests	
26.3.3 Algorithms and non-algorithms	
26.3.4 System tests	
26.3.5 Finding assumptions that do not hold	

## 26.1 What we want

Let's try a simple experiment. Write a binary search. Do it now. Don't wait until the end of the chapter. Don't wait until after the next section. It's important that you try. Now! A binary search is a search in a sorted sequence that starts at the middle:

- If the middle element is equal to what we are searching for, we are finished.
- If the middle element is less than what we are searching for, we look at the right-hand half, doing a binary search on that.
- If the middle element is greater than what we are searching for, we look at the left-hand half, doing a binary search on that.
- The result is an indicator of whether the search was successful and something that allows us to modify the element, if found, such as an index, a pointer, or an iterator.

Use less than (`<`) as the comparison (sorting) criterion. Feel free to use any data structure you like, any calling conventions you like, and any way of returning the result that you like, but do write the search code yourself. In this rare case, using someone else's function is counterproductive, even with proper acknowledgment. In particular, don't use the standard library algorithm (`binary_search` or `equal_range`) that would have been your first choice in most situations. Take as much time as you like.

So now you have written your binary search function. If not, go back to the previous paragraph. How do you know that your search function is correct? If you haven't already, write down why you are convinced that this code is correct. How confident are you about your reasoning? Are there parts of your argument that might be weak?

That was a trivially simple piece of code. It implemented a very regular and well-known algorithm. Your compiler is on the order of 200K lines of code, your operating system is 10M to 50M lines of code, and the safety-critical code in the airplane you'll fly on for your next vacation or conference is 500K to 2M lines of code. Does that make you feel comfortable? How do the techniques you used for your binary search function scale to real-world software sizes?

Curiously, given all that complex code, most software works correctly most of the time. We do not count anything running on a game-infested consumer PC as "critical." Even more importantly, safety-critical software works correctly just about all of the time. We cannot recall an example of a plane or a car crashing because of a software failure over the last decade. Stories about bank software getting seriously confused by a check for \$0.00 are now very old; such things essentially don't happen anymore. Yet software is written by people like you. You know that you make mistakes; we all do, so how do "they" get it right?

The most fundamental answer is that "we" have figured out how to build reliable systems out of unreliable parts. We try hard to make every program, every class, and every function correct, but we typically fail our first attempt at that. Then we debug, test, and redesign to find and remove as many errors as possible. However, in any nontrivial system, some bugs will still be hiding. We know that, but we can't find them — or rather, we can't find them all with the time and effort we are able and willing to expend. Then, we redesign the system yet again to recover from unexpected and "impossible" events. The result can be systems that are spectacularly reliable. Note that such reliable systems may still harbor errors — they usually do — and still occasionally work less well than we would like. However, they don't crash and always deliver minimally acceptable service. For example, a phone system may not manage to connect every call when demand is exceptionally high, but it never fails to connect many calls.

Now, we could be philosophical and discuss whether an unexpected error that we have conjectured and catered for is really an error, but let's not. It is more profitable and productive for systems builders "just" to figure how to make our systems more reliable.

### 26.1.1 Caveat

Testing is a huge topic. There are several schools of thought about how testing should be done, and different industries and application areas have different traditions and standards for testing. That's natural — you really don't need the same reliability standard for video games and avionics software — but it leads to confusing differences in terminology and tools. Treat this chapter as a source of ideas for your personal projects and as a source of ideals if you encounter testing of major systems. The testing of major systems involves a variety of combinations of tools and organizational structures that it would make little sense to try to describe here.

## 26.2 Proofs



Wait a minute! Why don't we just prove that our programs are correct, rather than fussing around with tests? As Edsger Dijkstra succinctly pointed out, "Testing can reveal the presence of errors, not their absence." This leads to an obvious desire to prove programs correct "much as mathematicians prove theorems."



Unfortunately, proving nontrivial programs correct is beyond the state of the art (outside very constrained applications domains), the proofs themselves can contain errors (as can the ones mathematicians produce), and the whole field of program proving is an advanced topic. So, we try as hard as we can to structure our programs so that we can reason about them and convince ourselves that they are correct. However, we also test (§26.3) and try to organize our code to be resilient against remaining errors (§26.4).

## 26.3 Testing

In §5.11, we described testing as "a systematic way to search for errors." Let's look at techniques for doing that.



People distinguish between *unit testing* and *system testing*. A "unit" is something like a function or a class that is a part of a complete program. If we test such units in isolation, we know where to look for the cause of problems when we find an error; any error will be in the unit that we are testing (or in the code we use to conduct the tests). This contrasts with system testing, where we test a complete system and all we know is that an error is "somewhere in the system." Typically, errors found in system testing – once we have done a good job at unit testing – relate to undesirable interactions between units. They are harder to find than errors within individual units and often more expensive to fix.

Obviously, a unit (say, a class) can be composed of other units (say, functions and other classes), and systems (say, an electronic commerce system) can be composed of other systems (say, a database, a GUI, a networking system, and an order validation system), so the distinction between unit testing and systems testing isn't as clear as you might have thought, but the general idea is that by testing our units well, we save ourselves work – and our end users pain.

One way of looking at testing is that any nontrivial system is built out of units, and these units are themselves built out of smaller units. So, we start testing the smallest units, then we test the units composed from those, and we work our way up until we have tested the whole system; that is, "the system" is just the largest unit (until we use that as a unit for some yet larger system).

So, let's first consider how to test a unit (such as a function, a class, a class hierarchy, or a template). Testers distinguish between white-box testing (where you can look at the detailed implementation of what you are testing) and black-box testing

(where you can look only at the interface of what you are testing). We will not make a big deal of this distinction; by all means read the implementation of what you test. But remember that someone might later come and rewrite that implementation, so try not to depend on anything that is not guaranteed in the interface. In fact, when testing anything, the basic idea is to throw anything we can at its interface to see if it responds reasonably.

Mentioning that someone (maybe yourself) might change the code after you tested it brings us to regression testing. Basically, whenever you make a change, you have to retest to make sure that you have not broken anything. So when you have improved a unit, you rerun its unit tests, and before you give the complete system to someone else (or use it for something real yourself), you run the complete system test.

Running such complete tests of a system is often called *regression testing* because it usually includes running tests that have previously found errors to see if these errors are still fixed. If not, the program has “regressed” and needs to be fixed again.

### 26.3.1 Regression tests

Building up a large collection of tests that have been useful for finding errors in the past is a major part of building an effective test suite for a system. Assume that you have users; they will send you bugs. Never throw away a bug report! Professionals use bug-tracking systems to ensure that. Anyway, a bug report demonstrates either an error in the system or an error in a user’s understanding of the system. Either way it is useful.

Usually, a bug report contains far too much extraneous information, and the first task of dealing with it is to produce the smallest program that exhibits the reported problem. This often involves cutting away most of the code submitted: in particular, we try to eliminate the use of libraries and application code that does not affect the error. Finding that minimal test program often helps us localize the bug in the system’s code, and that minimal program is what is added to the regression test suite. The way we find that minimal program is to keep removing code until the error disappears – and then reinsert the last bit of code we removed. This we do until we run out of candidates for removal.

Just running hundreds (or tens of thousands) of tests produced from old bug reports may not seem very systematic, but what we are really doing here is to systematically use the experience of users and developers. The regression test suite is a major part of a developer group’s institutional memory. For a large system, we simply can’t rely on having the original developers available to explain details of the design and implementation. The regression suite is what keeps a system from mutating away from what the developers and users have agreed to be its proper behavior.

### 26.3.2 Unit tests

OK. Enough words for now! Let's try a concrete example: let's test a binary search. Here is the specification from the ISO standard (§25.3.3.4):

```
template<class ForwardIterator, class T>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value);

template<class ForwardIterator, class T, class Compare>
bool binary_search(ForwardIterator first, ForwardIterator last,
                   const T& value, Compare comp);
```

**Requires:** The elements  $e$  of  $[first, last)$  are partitioned with respect to the expressions  $e < value$  and  $!(value < e)$  or  $comp(e, value)$  and  $!comp(value, e)$ . Also, for all elements  $e$  of  $[first, last)$ ,  $e < value$  implies  $!(value < e)$  or  $comp(e, value)$  implies  $!comp(value, e)$ .

**Returns:** true if there is an iterator  $i$  in the range  $[first, last)$  that satisfies the corresponding conditions:  $\!(\ast i < value) \&\& \!(value < \ast i)$  or  $comp(\ast i, value) == \text{false} \&\& comp(value, \ast i) == \text{false}$ .

**Complexity:** At most  $\log(last - first) + 2$  comparisons.

Nobody said that a formal specification (well, semiformal) was easy to read for the uninitiated. However, if you actually did the exercise of designing and implementing a binary search that we strongly suggested at the beginning of the chapter, you have a pretty good idea of what a binary search does and how to test it. This (standard) version takes a pair of forward iterators (§20.10.1) and a value as arguments and returns **true** if the value is in the range defined by the iterators. The iterators must define a sorted sequence. The comparison (sorting) criterion is **<**. We'll leave the second version of **binary\_search** that takes a comparison criterion as an extra argument as an exercise.

Here, we will deal only with errors that are not caught by the compiler, so examples like these are somebody else's problem:

```
binary_search(1,4,5);           // error: an int is not a forward iterator
vector<int> v(10);
binary_search(v.begin(),v.end(),"7"); // error: can't search for a string
                                         // in a vector of ints
binary_search(v.begin(),v.end());    // error: forgot the value
```

How can we *systematically* test **binary\_search()**? Obviously we can't just try every possible argument for it, because every possible argument would be every possible sequence of every possible type of value – that would be an infinite number of



tests! So, we must choose tests and to choose, we need some principles for making a choice:

- Test for *likely mistakes* (find the most errors).
- Test for *bad mistakes* (find the errors with the worst potential consequences).

By “bad,” we mean errors that would have the direst consequences. In general, that’s a fuzzy notion, but it can be made precise for a specific program. For example, for a binary search considered in isolation, all errors are about equally bad, but if we used that `binary_search()` in a program where all answers were carefully double-checked, getting a wrong answer from `binary_search()` might be far more acceptable than having it not return because it went into an infinite loop. In that case, we would spend greater effort tricking `binary_search()` into an infinite (or very long) loop than we would trying to trick it into giving a wrong answer. Note our use of “tricking” here. Testing is – among other things – an exercise in applying creative thinking to the problem of “How can we get this code to misbehave?” The best testers are not just systematic, but also quite devious (in a good cause, of course).

### 26.3.2.1 Testing strategy

How do we go about breaking `binary_search()`? We start by looking at `binary_search()`’s requirements, that is, what it assumes about its inputs. Unfortunately, from our perspective as testers, it is clearly stated that `[first,last)` must be a sorted sequence; that is, it is the caller’s job to ensure that, so we can’t fairly try to break `binary_search()` by giving it unsorted input or a `[first,last)` where `last < first`. Note that the requirements for `binary_search()` do not say what it will do if we give it input that doesn’t meet its requirements. Elsewhere in the standard, it says that it may throw an exception in that case, but it is not required to. These facts are good to remember for when we test uses of `binary_search()`, though, because a caller failing to establish the requirements of a function, such as `binary_search()`, is a likely source of errors.

We can imagine the following kinds of errors for `binary_search()`:

- Never returned (e.g., infinite loop)
- Crash (e.g., bad dereference, infinite recursion)
- Value not found even though it was in the sequence
- Value found even though it wasn’t in the sequence

In addition, we remember the following “opportunities” for user errors:

- The sequence is not sorted (e.g., `{2,1,5,-7,2,10}`).
- The sequence is not a valid sequence (e.g., `binary_search(&a[100], &a[50],77)`).

How might an implementer have made a mistake (for testers to find) for a simple call `binary_search(p1,p2,v)`? Errors often occur for “special cases.” In particular, when considering sequences (of any sort), we always look for the beginning and the end. In particular, the empty sequence should always be tested. So, let’s consider a few arrays of integers that are properly ordered as required:

<code>{ 1,2,3,5,8,13,21 }</code>	<i>// an “ordinary sequence”</i>
<code>{ }</code>	<i>// the empty sequence</i>
<code>{ 1 }</code>	<i>// just one element</i>
<code>{ 1,2,3,4 }</code>	<i>// even number of elements</i>
<code>{ 1,2,3,4,5 }</code>	<i>// odd number of elements</i>
<code>{ 1, 1, 1, 1, 1, 1, 1 }</code>	<i>// all elements equal</i>
<code>{ 0,1,1,1,1,1,1,1,1,1,1 }</code>	<i>// different element at beginning</i>
<code>{ 0,0,0,0,0,0,0,0,0,0,0,1 }</code>	<i>// different element at end</i>

Some test sequences are best generated by a program:

- `vector<int> v1;`
- `for (int i=0; i<1000000000; ++i) v.push_back(i); // a very large sequence`
- Some sequences with a random number of elements
- Some sequences with random elements (but still ordered)

This is not as systematic as we’d have liked. After all, we “just picked” some sequences. However, we used some fairly general rules of thumb that often are useful when dealing with sets of values; consider:

- The empty set
- Small sets
- Large sets
- Sets with extreme distributions
- Sets where “what is of interest” happens near the end
- Sets with duplicate elements
- Sets with even and with odd numbers of elements
- Sets generated using random numbers

We use the random sequences just to see if we can get lucky (i.e., find an error) with something we didn’t think about. It’s a brute-force technique, but relatively cheap in terms of our time.

Why “odd and even”? Well, lots of algorithms partition their input sequences, e.g., into the first half and the last half, and maybe the programmer considered only the odd or the even case. More generally, when we partition a sequence,

the point where we split it becomes the end of a subsequence, and we know that errors are likely near ends of sequences.

In general, we look for

- Extreme cases (large, small, strange distributions of input, etc.)
- Boundary conditions (anything near a limit)

What that really means, depends on the particular program we are testing.



### 26.3.2.2 A simple test harness

We have two categories of tests: tests that should succeed (e.g., searching for a value that's in a sequence) and tests that should fail (e.g., searching for a value in an empty sequence). For each of our sequences, let's construct some succeeding and some failing tests. We will start from the simplest and most obvious and proceed to improve until we have something that's good enough for our [binary\\_search](#) example:

```
vector<int> v { 1,2,3,5,8,13,21 };
if (binary_search(v.begin(),v.end(),1) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),5) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),8) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),21) == false) cout << "failed";
if (binary_search(v.begin(),v.end(),-7) == true) cout << "failed";
if (binary_search(v.begin(),v.end(),4) == true) cout << "failed";
if (binary_search(v.begin(),v.end(),22) == true) cout << "failed";
```

This is repetitive and tedious, but it will do for a start. In fact, many simple tests are nothing but a long list of calls like this. This naive approach has the virtue of being extremely simple. Even the newest member of the test team can add a new test to the set. However, we can usually do much better. For example, when something failed here, we are not told which test failed. That's unacceptable. Also, writing tests is no excuse for regressing to “cut and paste” programming. We need to consider the design of our testing code, just like any other code. So:

```
vector<int> v { 1,2,3,5,8,13,21 };
for (int x : {1,5,8,21,-7,2,44})
    if (binary_search(v.begin(),v.end(),x) == false) cout << x << " failed";
```

Assuming that we will eventually have dozens of tests, this will make a huge difference. For testing real-world systems, we often have many thousands of tests, so being precise about what test failed is essential.

Before going further, note another example of (semi-systematic) testing technique: we tested with correct values, choosing some from the ends of the sequence and some from “the middle.” For this sequence we could have tried all values, but typically that’s not a realistic option. For the failing values, we chose one from each end and one in the middle. Again, this is not perfectly systematic, but we begin to see a pattern that is useful whenever we deal with sequences of values or ranges of values – and that’s very common.

What’s wrong with these tests?

- We (initially) wrote the same things repeatedly.
- We (initially) numbered the tests manually.
- The output is very minimal (not very helpful).

After looking at this for a while, we decided to keep our tests as data in a file. Each test would contain an identifying label, a value to be looked up, a sequence, and an expected result. For example:

{ 27 7 { 1 2 3 5 8 13 21 } 0 }

This is test number **27**. It looks for **7** in the sequence **{ 1,2,3,5,8,13,21 }** expecting the result **0** (meaning **false**). Why do we put the test inputs in a file rather than placing them right into the text of the test program? Well, in this case we could have typed the tests straight into the program text, but having a lot of data in a source code file can be messy, and often, we use programs to generate test cases. Machine-generated test cases are typically in data files. Also, we can now write a test program that we can run with a variety of files of test cases:

```
struct Test {
    string label;
    int val;
    vector<int> seq;
    bool res;
};

istream& operator>>(istream& is, Test& t); // use the described format

int test_all(istream& is)
{
    int error_count = 0;
    for (Test t; is>>t; ) {
        bool r = binary_search(t.seq.begin(), t.seq.end(), t.val);
```

```

        if (r != t.res) {
            cout << "failure: test " << t.label
                << " binary_search: "
                << t.seq.size() << " elements, val==" << t.val
                << " -> " << t.res << '\n';
            ++error_count;
        }
    }
    return error_count;
}

int main()
{
    int errors = test_all(ifstream("my_tests.txt"));
    cout << "number of errors: " << errors << "\n";
}

```

Here is some test input using the sequences we listed above:

```

{1.11{123581321}1}
{1.25{123581321}1}
{1.38{123581321}1}
{1.421{123581321}1}
{1.5-7{123581321}0}
{1.64{123581321}0}
{1.722{123581321}0}

{21{}0}

{3.11{1}1}
{3.20{1}0}
{3.32{1}0}

```

Here we see why we used a string label rather than a number: that way we can “number” our tests using a more flexible system – here using a decimal system to indicate separate tests for the same sequence. A more sophisticated format would eliminate the need to repeat a sequence in our test data file.

### 26.3.2.3 Random sequences

When we choose values to be used in testing, we try to outwit the implementers (who are often ourselves) and to use values that focus on areas where we know bugs can hide (e.g., complicated sequences of conditions, the ends of sequences,

loops, etc.). However, that's also what we did when we tried to write and debug the code. So, we might repeat a logical mistake from the design when we design the tests and completely miss a problem. This is one reason it is a good idea to have someone different from the developer(s) involved with designing the tests. We have one technique that occasionally helps with that problem: just generate (a lot of) random values. For example, here is a function that writes a test description to `cout` using `randint()` from §24.7 and `std_lib_facilities.h`:

```
void make_test(const string& lab, int n, int base, int spread)
    // write a test description with the label lab to cout
    // generate a sequence of n elements starting at base
    // the average distance between elements is uniformly distributed
    // in [0:spread)
{
    cout << "{ " << lab << " " << n << " { ";
    vector<int> v;
    int elem = base;
    for (int i = 0; i < n; ++i) { // make elements
        elem += randint(spread);
        v.push_back(elem);
    }

    int val = base + randint(elem - base); // make search value
    bool found = false;
    for (int i = 0; i < n; ++i) { // print elements and see if val is found
        if (v[i] == val) found = true;
        cout << v[i] << " ";
    }
    cout << "}" << found << "\n";
}
```

Note that we did not use `binary_search` to see if the random `val` was in the random sequence. We can't use what we are testing to determine the correct value of a test.

Actually, `binary_search` isn't a particularly suitable example of the brute-force random number approach to testing. We doubt that this will find any bugs that are not picked up by our “hand-crafted” tests, but often this technique is useful. Anyway, let's make a few random tests:

```
int no_of_tests = randint(100); // make about 50 tests
for (int i = 0; i < no_of_tests; ++i) {
    string lab = "rand_test_";
    make_test(lab + to_string(i), // to_string from §23.2
```

```

randint(500),           // number of elements
0,                     // base
randint(50));          // spread
}

```

Generated tests based on random numbers are particularly useful when we need to test the cumulative effects of many operations where the result of an operation depends on how earlier operations were handled, that is, when a system has state; see §5.2.

The reason that random numbers are not all that useful for **binary\_search** is that each search of a sequence is independent of all other searches of that sequence. That of course assumes that the implementation of **binary\_search** hasn't done something terminally stupid, such as modifying its sequence. We have a better test for that (exercise 5).

### 26.3.3 Algorithms and non-algorithms

We have used **binary\_search()** as an example. It's a proper algorithm with

- Well-specified requirements on its inputs
- A well-specified effect on its inputs (in this case, no effects)
- No dependencies on objects that are not its explicit inputs
- Without serious constraints imposed by the environment (e.g., no specified time, space, or resource-sharing requirements)



It has obvious and explicitly stated pre- and post-conditions (§5.10). In other words, it's a tester's dream. Often, we are not so lucky: we have to test messy code that (at best) is defined by a somewhat sloppy English text and a couple of diagrams.

Wait a minute! Are we indulging in sloppy logic here? How can we talk about correctness and testing when we don't have a precise specification of what the code is supposed to do? The problem is that much of what needs to be done in software is not easy to specify in perfectly clear mathematical terms. Also, in many cases where it in theory could be specified like that, the math is beyond the abilities of the programmers who write and test the code. So we are left with the ideal of perfectly precise specifications and a reality of what someone (such as us) can manage under real-world conditions and time pressures.

So, assume that you have a messy function that you have to test. By "messy" we mean:

- *Inputs:* Its requirements on its (explicit or implicit) inputs are not specified quite as well as we would like.
- *Outputs:* Its (explicit or implicit) outputs are not specified quite as well as we would like.
- *Resources:* Its use of resources (time, memory, files, etc.) is not specified quite as well as we would like.

By “explicit or implicit” we mean that we have to look not just at the formal parameters and the return value, but also at any effects on global variables, **io-streams**, files, free-store memory allocation, etc. So, what can we do? First of all, such a function is almost certainly too long – or we could have stated its requirements and effects more clearly. Maybe we are talking about a function that is five pages long or uses “helper functions” in complicated and non-obvious ways. You may think that five pages is a lot for a function. It is, but we have seen much, much longer functions than that. Unfortunately, they are not uncommon.

If it is our code and if we had time, we would first of all try to break such a “messy function” up into smaller functions that come closer to our ideals of a well-specified function and first test those. However, here we will assume that our aim is to test the software – that is, to systematically find as many errors as possible – rather than (just) fixing bugs as we find them.

So, what do we look for? Our job as testers is to find errors. Where are bugs likely to hide? What characterizes code that is likely to contain bugs?

- Subtle dependencies on “other code”: look for use of global variables, non-**const**-reference arguments, pointers, etc.
- Resource management: look for memory management (**new** and **delete**), file use, locks, etc.
- Look for loops: check end conditions (as for **binary\_search()**).
- **if**-statements and switches (often referred to as “branching”): look for errors in their logic.

Let’s look at examples of each.

### 26.3.3.1 Dependencies

Consider this nonsense function:

```
int do_dependent(int a, int& b)      // messy function
    // undisciplined dependencies
{
    int val ;
    cin>>val;
    vec[val] += 10;
    cout << a;
    b++;
    return b;
}
```

To test **do\_dependent()**, we can’t just synthesize sets of arguments and see what it does with them. We have to take into account that it uses the global variables **cin**,

`cout`, and `vec`. That's pretty obvious in this little nonsense function, but in real code this may be hidden in a larger amount of code. Fortunately, there is software that can help us find such dependencies. Unfortunately, it is not always easily available or widely used. Assuming that we don't have analysis software to help us, we go through the function line by line, listing all its dependencies.

To test `do_dependent()`, we have to consider

- Its inputs:
  - The value of `a`
  - The value of `b` and the value of the `int` referenced by `b`
  - The input from `cin` (into `val`) and the state of `cin`
  - The state of `cout`
  - The value of `vec`, in particular, the value of `vec[val]`
- Its outputs:
  - The return value
  - The value of the `int` referenced by `b` (we incremented it)
  - The state of `cin` (beware of stream state and format state)
  - The state of `cout` (beware of stream state and format state)
  - The state of `vec` (we assigned to `vec[val]`)
  - Any exceptions that `vec` might have thrown (`vec[val]` might be out of range)

This is a long list. In fact, that list is longer than the function itself. This goes a long way toward explaining our dislike of global variables and our concerns about non-`const` references (and pointers). There really is something very nice about a function that just reads its arguments and produces a result as a return value: we can easily understand and test it.

Once the inputs and outputs are identified, we are basically back to the `binary_search()` case. We simply generate tests with input values (for explicit and implicit inputs) to see if they give the desired outputs (considering both implicit and explicit outputs). With `do_dependent()`, we would probably start with a very large `val` and a negative `val`, to see what happens. It looks as if `vec` had better be a range-checked `vector` (or we can very simply generate really bad errors). We would of course check what the documentation said about all those inputs and outputs, but with a messy function like that we have little hope of the specification being complete and precise, so we will just break the functions (i.e., find errors) and start asking questions about what is correct. Often, such testing and questions should lead to a redesign.

### 26.3.3.2 Resource management

Consider this nonsense function:

```
void do_resources1(int a, int b, const char* s) // messy function
    // undisciplined resource use
{
    FILE* f = fopen(s, "r");
    // open file (C style)
    int* p = new int[a];
    // allocate some memory
    if (b<=0) throw Bad_arg();
    // maybe throw an exception
    int* q = new int[b];
    // allocate some more memory
    delete[] p;
    // deallocate the memory pointed to by p
}
```

To test **do\_resources1()**, we have to consider whether every resource acquired has been properly disposed of, that is, whether every resource has been either released or passed to some other function.

Here, it is obvious that

- The file named **s** is not closed
- The memory allocated for **p** is leaked if **b<=0** or if the second **new** throws
- The memory for **q** is leaked if **0<b**

In addition, we should always consider the possibility that an attempt at opening a file might fail. To get this miserable result, we deliberately used a very old-fashioned programming style (**fopen()** is the standard C way of opening files). We could have made the job for testers more straightforward by writing

```
void do_resources2(int a, int b, const char* s) // less messy function
{
    ifstream is(s);
    // open file
    vector<int> v1(a);
    // create vector (owning memory)
    if (b<=0) throw Bad_arg();
    // maybe throw an exception
    vector<int> v2(b);
    // create another vector (owning memory)
}
```

 Now every resource is owned by an object with a destructor that will release it. Considering how we could write a function more simply (more cleanly) is sometimes a good way to get ideas for testing. The “Resource Acquisition Is Initialization” (RAII) technique from §19.5.2 provides a general strategy for this kind of resource management problem.

 Please note that resource management is not just checking that every piece of memory allocated is deleted. Sometimes we receive resources from elsewhere (e.g.,

as an argument), and sometimes we pass resources out of a function (e.g., as a return value). It can be quite hard to determine what is right about such cases. Consider:

```
FILE* do_resources3(int a, int* p, const char* s)      // messy function
    // undisciplined resource passing
{
    FILE* f = fopen(s,"r");
    delete p;
    delete var;
    var = new int[27];
    return f;
}
```

Is it right for **do\_resources3()** to pass the (supposedly) opened file back as the return value? Is it right for **do\_resources3()** to delete the memory passed to it as the argument **p**? We also added a really sneaky use of the global variable **var** (obviously a pointer). Basically, passing resources in and out of functions is common and useful, but to know if it is correct requires knowledge of a resource management strategy. Who owns the resource? Who is supposed to delete/release it? The documentation should clearly and simply answer those questions. (Dream on.) In either case, passing of resources is a fertile area for bugs and a tempting target for testing.

Note how we (deliberately) complicated the resource management example by using a global variable. Things can get really messy when we start to mix the sources of likely bugs. As programmers, we try to avoid that. As testers, we look for such examples as easy pickings.

### 26.3.3.3 Loops

We have looked at loops when we discussed **binary\_search()**. Basically most errors occur at the ends:

- Is everything properly initialized when we start the loop?
- Do we correctly end with the last case (often the last element)?

Here is an example where we get it wrong:

```
int do_loop(const vector<int>& v)      // messy function
    // undisciplined loop
{
    int i;
    int sum;
    while(i<=vec.size()) sum+=v[i];
    return sum;
}
```

There are three obvious errors. (What are they?) In addition, a good tester will immediately spot the opportunity for an overflow where we are adding to **sum**:

- Many loops involve data and might cause some sort of overflow when they are given large inputs.

A famous and particularly nasty loop error, the buffer overflow, falls into the category that can be caught by systematically asking the two key questions about loops:

```
char buf[MAX];      // fixed-size buffer

char* read_line()    // dangerously sloppy
{
    int i = 0;
    char ch;
    while(cin.get(ch) && ch != '\n') buf[i++] = ch;
    buf[i+1] = 0;
    return buf;
}
```

Of course, *you* wouldn't write something like that! (Why not? What's so wrong with **read\_line()**?) However, it is sadly common and comes in many variations, such as

```
// dangerously sloppy:
gets(buf);           // read a line into buf
scanf("%s",buf);    // read a line into buf
```

Look up **gets()** and **scanf()** in your documentation and avoid them like the plague. By "dangerous," we mean that such buffer overflows are a staple of "cracking" – that is, break-ins – on computers. Many implementations now warn against **gets()** and its cousins for exactly this reason.

#### 26.3.3.4 Branching

Obviously, when we have to make a choice, we may make the wrong choice. This makes **if**-statements and **switch**-statements good targets for testers. There are two major problems to look for:

- Are all possibilities covered?
- Are the right actions associated with the right possibilities?

Consider this nonsense function:

```
void do_branch1(int x, int y)      // messy function
    // undisciplined use of if
{
    if (x<0) {
        if (y<0)
            cout << "very negative\n";
        else
            cout << "somewhat negative\n";
    }
    else if (x>0) {
        if (y<0)
            cout << "very positive\n";
        else
            cout << "somewhat positive\n";
    }
}
```

The most obvious error here is that we “forgot” the case where **x** is 0. When testing against zero (or for positive and negative values), zero is often forgotten or lumped with the wrong case (e.g., considered negative). Also, there is a more subtle (but not uncommon) error lurking here: the actions for (**x>0 && y<0**) and (**x>0 && y>=0**) have “somehow” been reversed. This happens a lot with cut-and-paste editing.

The more complicated the use of **if**-statements is, the more likely such errors become. From a tester’s point of view, we look at such code and try to make sure that every branch is tested. For **do\_branch1()** the obvious test set is

```
do_branch1(-1,-1);
do_branch1(-1, 1);
do_branch1(1,-1);
do_branch1(1,1);
do_branch1(-1,0);
do_branch1(0,-1);
do_branch1(1,0);
do_branch1(0,1);
do_branch1(0,0);
```

Basically, that’s the brute-force “try all the alternatives” approach after we noticed that **do\_branch1()** tested against 0 using **<** and **>**. To catch the wrong actions for positive values of **x**, we have to combine the calls with their desired output.

Dealing with **switch**-statements is fundamentally similar to dealing with **if**-statements.

```
void do_branch1(int x, int y) // messy function
    // undisciplined use of switch
{
    if (y<0 && y<=3)
        switch (x) {
            case 1:
                cout << "one\n";
                break;
            case 2:
                cout << "two\n";
            case 3:
                cout << "three\n";
        }
}
```

Here we have made four classic mistakes:

- We range checked the wrong variable (**y** instead of **x**).
- We forgot a break statement leading to a wrong action for **x==2**.
- We forgot a default case (thinking we had taken care of that with the **if**-statement).
- We wrote **y<0** when we meant to say **0<y**.



As testers, we always look for unhandled cases. Please note that “just fixing the problem” is not enough. It may reappear when we are not looking. As testers, we want to write tests that systematically catch errors. If we just fixed this simple code, we may very well get our fix wrong so that it either doesn’t solve the problem or introduces new and different errors. The purpose of looking at the code is not really to spot errors (though that’s always useful), but to design a suitable set of tests that will catch all errors (or, more realistically, will catch many errors).



Note that loops have an implicit “**if**”: they test whether we have reached the end. Thus loops are also branching statements. When we look at programs containing branching, the first question is always, “Have we covered (tested) every branch?” Surprisingly that is not always possible in real code (because in real code, a function is called as needed by other functions and not necessarily in all possible ways). Consequently, a common question for testers is, “What is your code coverage?” and the answer had better be, “We tested most branches,” followed by an explanation of why the remaining branches are hard to reach. 100% coverage is the ideal.

### 26.3.4 System tests

Testing any significant system is a skilled job. For example, the testing of the computers that control telephone systems takes place in specially constructed rooms with racks full of computers simulating the traffic of tens of thousands of people. Such systems cost millions and are the work of teams of very skilled engineers. After it is deployed, a main telephone switch is supposed to work continuously for 20 years with at most 20 minutes of downtime (for any reason, including power failures, flooding, and earthquakes). We will not go into detail here – it would be easier to teach a physics freshman to calculate course corrections for a Mars probe – but we'll try to give you some ideas that could be useful for a smaller project or for understanding the testing of a larger system.

First of all, please remember that the purpose of testing is to find errors, especially potentially frequent and potentially serious errors. It is not simply to write and run the largest number of tests. This implies that some understanding of the system being tested is highly desirable. Even more than for unit testing, effective system testing relies on knowledge of the application (domain knowledge). Developing a system takes more than just knowledge of programming language issues and computer science; it requires an understanding of the application areas and of the people who use the applications. This is something we find important for motivating us to work with code: we get to see so many interesting applications and meet interesting people.

For a complete system to be tested, it has to be built out of all of its parts (units). This can take significant time, so many system tests are run just once a day (often at night while the developers are supposed to be asleep) after all unit tests have been done. Regression tests are a key component here. The areas of a program in which we are most likely to find errors are new code and areas of code where errors were found earlier. So running the collection of old tests (the regression tests) is essential; without those a large system will never become stable. We would introduce new bugs as fast as we removed old ones.

Note that we take it for granted that when we fix a few errors, we accidentally introduce a few new ones. We hope the number of new bugs is lower than the number of old ones that we removed, and that the consequences of the new ones are less severe. However, at least until we have rerun our regression tests and added new tests for our new code, we must assume that our system is broken (by our bug fixes).

### 26.3.5 Finding assumptions that do not hold

The specification of `binary_search` clearly stated that the sequence in which we search must be sorted. That deprived us of many opportunities for sneaky unit tests. But obviously there are opportunities for writing bad code that we have not devised tests to detect (except for the system tests). Can we use our understanding of a system's “units” (functions, classes, etc.) to devise better tests?

Unfortunately, the simplest answer is no. As pure testers, we cannot change the code, but to detect violations of an interface's requirements (pre-conditions), someone must either check before each call or as part of the implementation of each call (see §5.5). However, if we are testing our own code, we can insert such tests. If we are testers and the people who write the code will listen to us (that's not always the case), we can tell them about the unchecked requirements and have them ensure that they are checked.

Consider again **binary\_search**: we couldn't test that the input sequence **[first:last]** really was a sequence and that it was sorted (§26.3.2.2). However, we could write a function that does check:

```
template<class Iter, class T>
bool b2(Iter first, Iter last, const T& value)
{
    // check if [first:last) is a sequence:
    if (last<first) throw Bad_sequence();

    // check if the sequence is ordered:
    if (2<=last-first)
        for (Iter p = first+1; p<last; ++p)
            if (*p<*(p-1)) throw Not_ordered();

    // all's OK, call binary_search:
    return binary_search(first,last,value);
}
```

Now, there are reasons why **binary\_search** isn't written with such tests, including these:

- The test for **last<first** can't be done for a forward iterator; for example, the iterator for **std::list** does not have a **<** (§B.3.2). In general, there is no really good way of testing that a pair of iterators defines a sequence (starting to iterate from **first** hoping to meet **last** is not a good idea).
- Scanning the sequence to check that the values are ordered is far more expensive than executing **binary\_search** itself (the real purpose of **binary\_search** is not to have to blindly walk through the sequence looking for a value the way **std::find** does).

So what could we do? We could replace **binary\_search** with **b2** when we are testing (only for calls to **binary\_search** with random-access iterators, though).

Alternatively, we could have the implementer of **binary\_search** insert code that a tester could enable:

```
template<class Iter, class T>      // warning: contains pseudo code
bool binary_search (Iter first, Iter last, const T& value)
{
    if (test enabled) {
        if (Iter is a random access iterator) {
            // check if [first:last) is a sequence:
            if (last<first) throw Bad_sequence();
        }

        // check if the sequence is ordered:
        if (first!=last) {
            Iter prev = first;
            for (Iter p = ++first; p!=last; ++p, ++prev)
                if (*p<*prev) throw Not_ordered();
        }
    }

    // now do binary_search
}
```

Since the meaning of **test enabled** depends on how testing of code is arranged (for a specific system in a specific organization), we have left it as pseudo code: when testing your own code, you could simply have a **test\_enabled** variable. We also left the **Iter is a random access iterator** test as pseudo code because we haven't explained "iterator traits." Should you really need such a test, look up *iterator traits* in a more advanced C++ textbook.

## 26.4 Design for testing

When we start writing a program, we know that we would like it to eventually be complete and correct. We also know that to achieve that, we must test it. Consequently, we try to design for correctness and testing from day one. In fact, many good programmers have as their slogan "Test early and often" and don't write any code before they have some idea about how they would go about testing it. Thinking about testing early helps to avoid errors in the first place (as well as helping to find them later). We subscribe to that philosophy. Some programmers even write unit tests before they implement a unit.



The example in §26.3.2.1 and the examples in §26.3.3 illustrate these key notions:

- Use well-defined interfaces so that you can write tests for the use of these interfaces.
- Have a way of representing operations as text so that they can be stored, analyzed, and replayed. This also applies to output operations.
- Embed tests of otherwise unchecked assumptions (assertions) in the calling code to catch bad arguments before system testing.
- Minimize dependencies and keep dependencies explicit.
- Have a clear resource management strategy.

Philosophically, this could be seen as enabling unit-testing techniques for subsystems and complete systems.

If performance didn't matter, we could leave the test of the (otherwise) unchecked assumptions (requirements, pre-conditions) enabled all the time. However, there are usually reasons why they are not systematically checked. For example, we saw how checking whether a sequence is sorted is both complicated and far more expensive than using **binary\_sort**. Consequently, it is a good idea to design a system that allows us to selectively enable and disable such checks. For many systems, it is a good idea to leave a fair number of the cheaper checks enabled even in the final (shipping) version: sometimes "impossible" things happen and we would prefer to know about them from a specific error message rather than from a simple crash.

## 26.5 Debugging

Debugging is an issue of technique and attitude. Of these, attitude is the more important. Please revisit Chapter 5. Note how debugging and testing differ. Both catch bugs, but debugging is much more ad hoc and typically concerned with removing known bugs and implementing features. Whatever we can do to make debugging more like testing should be done. It is a slight exaggeration to say that we love testing, but we definitely hate debugging. Good early unit testing and design for testing help minimize debugging.

## 26.6 Performance

Having a program correct is not enough for it to be useful. Even assuming that it has sufficient facilities to make it useful, it must also provide appropriate performance. A good program is "efficient enough"; that is, it will run in an acceptable time given the resources available. Note that absolute efficiency is uninteresting,

and an obsession with getting a program to run fast can seriously damage development by complicating code (leading to more bugs and more debugging) and making maintenance (including porting and performance tuning) more difficult and costly.

So, how can we know that a program (or a unit of a program) is “efficient enough”? In the abstract we cannot know, and for many programs the hardware is so fast that the question doesn’t arise. We have seen products shipped that were compiled in debug mode (i.e., running about 25 times slower than necessary) to enable better diagnostics for errors occurring after deployment (this can happen to even the best code when it has to coexist with code developed “elsewhere”).

Consequently, the answer to the “Is it efficient enough?” question is: “Measure how long interesting test cases take.” To do that, you obviously have to know your end users well enough to have an idea of what they would consider “interesting” and how much time such interesting uses can acceptably take. Logically, we simply clock our tests with a stopwatch and check that none consumes an unreasonable amount of time. This becomes practical when we use facilities such as **system\_clock** (§26.6.1) to do the timing for us, and we can automatically compare the time taken by tests with estimates of what is reasonable. Alternatively (or additionally) we can record how long tests take and compare them to earlier test runs. This way we get a form of regression test for performance.

Some of the worst performance bugs are caused by poor algorithms and can be found by testing. One reason for testing with large sets of data is to expose inefficient algorithms. As an example, assume that an application has to make sums of the elements in rows of a matrix (using the **Matrix** library from Chapter 24). Someone supplied an appropriate function:

```
double row_sum(Matrix<double,2> m, int n); // sum of elements in m[n]
```

Now someone uses that to generate a **vector** of sums where **v[n]** is the sum of the elements of the first **n** rows:

```
double row_accum(Matrix<double,2> m, int n) // sum of elements in m[0:n)
{
    double s = 0;
    for (int i=0; i<n; ++i) s+=row_sum(m,i);
    return s;
}

// compute accumulated sums of rows of m:
vector<double> v;
for (int i = 0; i<m.dim1(); ++i) v.push_back(row_accum(m,i+1));
```

You can imagine this to be part of a unit test or executed as part of the application exercised by a system test. In either case, you will notice something strange if the matrix ever gets really large: basically, the time needed goes up with the square of the size of **m**. Why? What we did was to add all the elements of the first row, then we added all the elements in the second row (revisiting all the elements of the first row), then we added all the elements in the third row (revisiting all the elements of the first and second rows), etc.

If you think this example was bad, consider what would have happened if the **row\_sum()** had had to access a database to get its data. Reading from disk is many thousands of times slower than reading from main memory.

Now, you may complain: “Nobody would write something that stupid!” Sorry, but we have seen much worse, and usually a poor algorithm (from the performance point of view) is not that easy to spot when buried in application code. Did you spot the performance problem when you first glanced at the code? A problem can be quite hard to spot unless you are specifically looking for that particular kind of problem. Here is a simple real-world example found in a server:



```
for (int i=0; i<strlen(s); ++i) {
    // ... do something with s[i] ...
}
```

Often, **s** was a string with about 20K characters.

Not all performance problems have to do with poor algorithms. In fact (as we pointed out in §26.3.3), much of the code we write cannot be classified as proper algorithms. Such “non-algorithmic” performance problems typically fall under the broad classification of “poor design.” They include

- Repeated recalculation of information (e.g., the row-summing problem above)
- Repeated checking of the same fact (e.g., checking that an index is in range each time it is used in a loop or checking an argument repeatedly as it is passed unchanged from function to function)
- Repeated visits to the disk (or to the web)

Note the (repeated) *repeated*. Obviously, we mean “unnecessarily repeated,” but the point is that unless you do something many times, it will not have an impact on performance. We are all for thorough checking of function arguments and loop variables, but if we do the same check a million times for the same values, those redundant checks just might hurt performance. If we – by measurement – find that performance is hurt, we will try to see if we can remove a repeated action. Don’t do that unless you are sure that performance is really a problem. Premature optimization is the source of many bugs and much wasted time.

### 26.6.1 Timing

How do you know if a piece of code is fast enough? How do you know how long an operation takes? Well, in many cases where it matters, you can simply look at a clock (stopwatch, wall clock, or wristwatch). That's not scientific or accurate, but if that's not feasible, you can often conclude that the program was fast enough. It is not good to be obsessed with performance.

If you need to measure smaller increments of time or if you can't sit around with a stopwatch, you need to get your computer to help you; it knows the time and can give it to you. For example, on a Unix system, simply prefixing a command with **time** will make the system print out the time taken. You might use **time** to figure out how long it takes to compile a C++ source file **x.cpp**. Normally, you compile it like this:

```
g++ x.cpp
```

To get that compilation timed, you just add **time**:

```
time g++ x.cpp
```

This will compile **x.cpp** and also print the time taken on the screen. This is a simple and effective way of timing small programs. Remember to always do several timing runs because “other activities” on your machine might interfere. If you get roughly the same answer three times, you can usually trust the result.

But what if you want to measure something that takes just milliseconds? What if you want to do your own, more detailed, measurements of a part of a program? You use standard library facilities from **<chrono>**. For example, to measure the time used by a function **do\_something()** you can write code like this:

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

int main()
{
    int n = 10000000;           // repeat do_something() n times

    auto t1 = system_clock::now(); // begin time

    for (int i = 0; i<n; i++) do_something(); // timing loop
```

```
auto t2 = system_clock::now();           // end time

cout << "do_something() " << n << " times took "
<< duration_cast<milliseconds>(t2-t1).count() << "milliseconds\n";
}
```

The `system_clock` is one of the standard timers, and `system_clock::now()` returns the point of time (a `time_point`) at which it is called. Subtract two `time_points` (here, `t2-t1`) and you get a length of time (a `duration`). We can use `auto` to save us from the details of the `duration` and `time_point` types, which are surprisingly complicated if your view of time is simply what you see on a wristwatch. In fact, the standard library's timing facilities were originally designed for advanced physics applications and are far more flexible and general than most users need.

To get a `duration` in terms of a particular unit of time, such as `seconds`, `milliseconds`, or `nanoseconds`, we convert ("cast") it to that unit using the conversion function `duration_cast`. You need something like `duration_cast` because different systems and different clocks measure time in different units. Don't forget the `.count()`. That is what extracts the number of units ("clock ticks") from the `duration` that contains both the clock ticks and their unit.

The `system_clock` is meant to measure intervals from a fraction of a second to a few seconds. Don't try to use it to measure hours.

Again, don't believe any time measurement that you cannot repeat with roughly the same result three times. What does "roughly the same" mean? "Within 10%" is a reasonable answer. Remember that modern computers are *fast*: 1,000,000,000 instructions per second is ordinary. This implies that you won't be able to measure anything unless you can repeat it tens of thousands of times or it does something really slow, such as writing to disk or accessing the web. In the latter case, you just have to get it to repeat a few hundred times, but you have to worry that so much is going on that you might not understand the results.

## 26.7 References

- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.  
Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2003. ISBN 0321194330.



## Drill

Get the test of **binary\_search** to run:

1. Implement the input operator for **Test** from §26.3.2.2.
2. Complete a file of tests for the sequences from §26.3:
  - a. `{ 1 2 3 5 8 13 21 }` *// an “ordinary sequence”*
  - b. `{ }`
  - c. `{ 1 }`
  - d. `{ 1 2 3 4 }` *// even number of elements*
  - e. `{ 1 2 3 4 5 }` *// odd number of elements*
  - f. `{ 1 1 1 1 1 1 1 }` *// all elements equal*
  - g. `{ 0 1 1 1 1 1 1 1 1 1 1 }` *// different element at beginning*
  - h. `{ 0 0 0 0 0 0 0 0 0 0 1 }` *// different element at end*
3. Based on §26.3.1.3, complete a program that generates
  - a. A very large sequence (what would you consider very large, and why?)
  - b. Ten sequences with a random number of elements
  - c. Ten sequences with 0, 1, 2 . . . 9 random elements (but still ordered)
4. Repeat these tests for sequences of strings, such as `{ Bohr Darwin Einstein Lavoisier Newton Turing }`.

## Review

1. Make a list of applications, each with a brief explanation of the worst thing that can happen if there is a bug; e.g., airplane control – crash: 231 people dead; \$500M equipment loss.
2. Why don’t we just prove our programs correct?
3. What’s the difference between unit testing and system testing?
4. What is regression testing and why is it important?
5. What is the purpose of testing?
6. Why doesn’t **binary\_search** just check its requirements?
7. If we can’t check for all possible errors, what kinds of errors do we primarily look for?
8. Where are bugs most likely to occur in code manipulating a sequence of elements?
9. Why is it a good idea to test for large values?
10. Why do we often represent tests as data rather than as code?
11. Why and when would we use lots of tests based on random values?

12. Why is it hard to test a program using a GUI?
13. What is needed to test a “unit” in isolation?
14. What is the connection between testability and portability?
15. What makes testing a class harder than testing a function?
16. Why is it important that tests be repeatable?
17. What can a tester do when finding that a “unit” relies on unchecked assumptions (pre-conditions)?
18. What can a designer/implementer do to improve testing?
19. How does testing differ from debugging?
20. When does performance matter?
21. Give two (or more) examples of how to (easily) create bad performance problems.

## Terms

assumptions	pre-condition	test coverage
black-box testing	proof	test harness
branching	regression	testing
design for testing	resource usage	timing
inputs	state	unit test
outputs	<b>system_clock</b>	white-box testing
post-condition	system test	

## Exercises

1. Run your **binary search** algorithm from §26.1 with the tests presented in §26.3.2.1.
2. Modify the testing of **binary\_search** to deal with arbitrary element types. Then, test it with **string** sequences and floating-point sequences.
3. Repeat exercise 1 with the version of **binary\_search** that takes a comparison criterion. Make a list of new opportunities for errors introduced by that extra argument.
4. Devise a format for test data so that you can define a sequence once and run several tests against it.
5. Add a test to the set of **binary\_search** tests to try to catch the (unlikely) error of a **binary\_search** modifying the sequence.
6. Modify the calculator from Chapter 7 minimally to let it take input from a file and produce output to a file (or use your operating system’s facilities for redirecting I/O). Then devise a reasonably comprehensive test for it.
7. Test the “simple text editor” from §20.6.

8. Add a text-based interface to the graphics interface library from Chapters 12–15. For example, the string `Circle(Point(0,1),15)` should generate a call `Circle(Point(0,1),15)`. Use this text interface to make a “kid’s drawing” of a two-dimensional house with a roof, two windows, and a door.
9. Add a text-based output format for the graphics interface library. For example, when a call `Circle(Point(0,1),15)` is executed, a string like `Circle(Point(0,1),15)` should be produced on an output stream.
10. Use the text-based interface from exercise 9 to write a better test for the graphical interface library.
11. Time the sum example from §26.6 with `m` being square matrices with dimensions 100, 10,000, 1,000,000, and 10,000,000. Use random element values in the range [-10:10). Rewrite the calculation of `v` to use a more efficient (not  $O(N^2)$ ) algorithm and compare the timings.
12. Write a program that generates random floating-point numbers and sort them using `std::sort()`. Measure the time used to sort 500,000 `doubles` and 5,000,000 `doubles`.
13. Repeat the experiment in the previous exercise, but with random strings of lengths in the [0:100) range.
14. Repeat the previous exercise, except using a `map` rather than a `vector` so that we don’t need to sort.

## Postscript

As programmers, we dream about writing beautiful programs that just work – preferably the first time we try them. The reality is different: it is hard to get programs right, and it is hard to get them to stay right as we (and our colleagues) work to improve them. Testing – including design for testing – is a major way of ensuring that the systems we ship actually work. Whenever we reach the end of a day in our highly technological world, we really ought to give a kind thought to the (often forgotten) testers.





# The C Programming Language

“C is a strongly typed,  
weakly checked,  
programming language.”

—Dennis Ritchie

This chapter is a brief overview of the C programming language and its standard library from the point of view of someone who knows C++. It lists the C++ features missing from C and gives examples of how a C programmer can cope without those. C/C++ incompatibilities are presented, and C/C++ interoperability is discussed. Examples of I/O, list manipulation, memory management, and string manipulation are included as illustration.

## 27.1 C and C++: siblings

- 27.1.1 C/C++ compatibility
- 27.1.2 C++ features missing from C
- 27.1.3 The C standard library

## 27.2 Functions

- 27.2.1 No function name  
overloading
- 27.2.2 Function argument type  
checking
- 27.2.3 Function definitions
- 27.2.4 Calling C from C++ and C++  
from C
- 27.2.5 Pointers to functions

## 27.3 Minor language differences

- 27.3.1 `struct` tag namespace
- 27.3.2 Keywords
- 27.3.3 Definitions
- 27.3.4 C-style casts
- 27.3.5 Conversion of `void*`
- 27.3.6 `enum`
- 27.3.7 Namespaces

## 27.4 Free store

### 27.5 C-style strings

- 27.5.1 C-style strings and `const`
- 27.5.2 Byte operations
- 27.5.3 An example: `strcpy()`
- 27.5.4 A style issue

## 27.6 Input/output: stdio

- 27.6.1 Output
- 27.6.2 Input
- 27.6.3 Files

## 27.7 Constants and macros

## 27.8 Macros

- 27.8.1 Function-like macros
- 27.8.2 Syntax macros
- 27.8.3 Conditional compilation

## 27.9 An example: intrusive containers

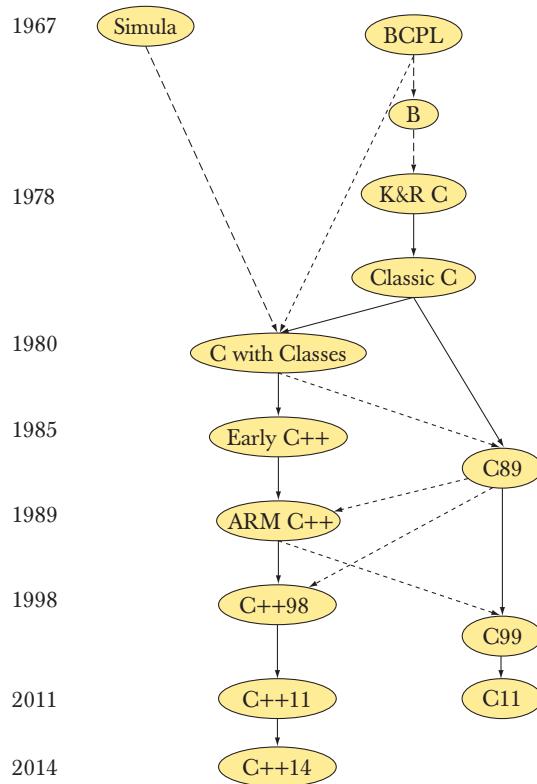
## 27.1 C and C++: siblings

The C programming language was designed and implemented by Dennis Ritchie at Bell Labs and popularized by the book *The C Programming Language* by Brian Kernighan and Dennis Ritchie (colloquially known as “K&R”), which is arguably still the best introduction to C and one of the great books on programming (§22.2.5). The text of the original definition of C++ was an edit of the text of the 1980 definition of C, supplied by Dennis Ritchie. After this initial branch, both languages evolved further. Like C++, C is now defined by an ISO standard.

We see C primarily as a subset of C++. Thus, from a C++ point of view, the problem of describing C boils down to two issues:

- Describe where C isn’t a subset of C++.
- Describe which C++ features are missing in C and which facilities and techniques can be used to compensate.

Historically, modern C and modern C++ are siblings. Both are direct descendants of “Classic C,” the dialect of C popularized by the first edition of Kernighan and Ritchie’s *The C Programming Language* plus structure assignment and enumerations:



The version of C that is used today is still mostly C89 (as described in the second edition of K&R), and that's what we are describing here. There is still some Classic C in use and some C99, but that should not cause you any problems when you know C++ and C89.

Both C and C++ were “born” in the Computer Science Research Center of Bell Labs in Murray Hill, New Jersey (for a while, my office was a couple of doors down and across the corridor from those of Dennis Ritchie and Brian Kernighan):





Both languages are now defined/controlled by ISO standards committees. For each, many supported implementations are in use. Often, an implementation supports both languages with the desired language chosen by a compiler switch or a source file suffix. Both are available on more platforms than any other language. Both were primarily designed for and are now heavily used for hard system programming tasks, such as

- Operating system kernels
- Device drivers
- Embedded systems
- Compilers
- Communications systems

There are no performance differences between equivalent C and C++ programs.

Like C++, C is very widely used. Taken together, the C/C++ community is the largest software development community on earth.

### 27.1.1 C/C++ compatibility



It is not uncommon to hear references to “C/C++.” However, there is no such language, and the use of “C/C++” is typically a sign of ignorance. We use “C/C++” only in the context of C/C++ compatibility issues and when talking about the large shared C/C++ technical community.



C++ is largely, but not completely, a superset of C. With a few very rare exceptions, constructs that are both C and C++ have the same meaning (semantics) in both languages. C++ was designed to be “as close as possible to C, but no closer”:

- For ease of transition
- For coexistence

Most incompatibilities relate to C++’s stricter type checking.

An example of a program that is legal C but not C++ is one that uses a C++ keyword that is not a C keyword as an identifier (see §27.3.2):

```
int class(int new, int bool); /* C, but not C++ */
```

Examples where the semantics differ for a construct that is legal in both languages are harder to find, but here is one:

```
int s = sizeof('a'); /* sizeof(int), often 4 in C and 1 in C++ */
```

The type of a character literal, such as '`a`', is `int` in C and `char` in C++. However, for a `char` variable `ch` we have `sizeof(ch)==1` in both languages.

Information related to compatibility and language differences is not exactly exciting. There are no new neat programming techniques to learn. You might like `printf()` (§27.6), but with that possible exception (and some feeble attempts at geek humor), this chapter is bone dry. Its purpose is simple: to allow you to read and write C if you need to. This includes pointing out the hazards that are obvious to experienced C programmers, but typically unexpected by C++ programmers. We hope you can learn to avoid those hazards with minimal grief.

Most C++ programmers will have to deal with C code at some point or another, just as most C programmers will have to deal with C++ code. Much of what we describe in this chapter will be familiar to most C programmers, but some will be considered “expert level.” The reason for that is simple: not everyone agrees about what is “expert level” and we just describe what is common in real-world code. Maybe understanding compatibility issues can be a cheap way of gaining an unfair reputation as a “C expert.” But do remember: real expertise is in the use of a language (in this case C), rather than in understanding esoteric language rules (as are exposed by considering compatibility issues).

## References

- ISO/IEC 9899:1999. *Programming Languages – C*. This defines C99; most implementations implement C89 (often with a few extensions).
- ISO/IEC 9899:2011. *Programming Languages – C*. This defines C11.
- ISO/IEC 14882:2011. *Programming Languages – C++*.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, 1988. ISBN 0131103628.
- Stroustrup, Bjarne. “Learning Standard C++ as a New Language.” *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. “C and C++: Siblings”; “C and C++: A Case for Compatibility”; and “C and C++: Case Studies in Compatibility.” *The C/C++ Users Journal*, July, Aug., and Sept. 2002.

The papers by Stroustrup are most easily found on my publications home page.

### 27.1.2 C++ features missing from C

From a C++ perspective, C (i.e., C89) lacks a lot of features, such as

- Classes and member functions
- Use `struct` and global functions.

- Derived classes and virtual functions
  - Use **structs**, global functions, and pointers to functions (§27.2.3).
- Templates and inline functions
  - Use macros (§27.8).
- Exceptions
  - Use error codes, error return values, etc.
- Function overloading
  - Give each function a distinct name.
- **new/delete**
  - Use **malloc()/free()** and separate initialization/cleanup code.
- References
  - Use pointers.
- **const, constexpr**, or functions in constant expressions
  - Use macros.
- **bool**
  - Use **int**.
- **static\_cast, reinterpret\_cast**, and **const\_cast**
  - Use C-style casts, e.g., **(int)a** rather than **static<int>(a)**.



Lots of useful code is written in C, so this list should remind us that no one language feature is absolutely necessary. Most language features – even most C language features – are there for the convenience (only) of the programmer. After all, given sufficient time, cleverness, and patience, every program can be written in assembler. Note that because C and C++ share a machine model that is very close to the real machine, they are well suited to emulate varieties of programming styles.

The rest of this chapter explains how to write useful programs without those features. Our basic advice for using C is:

- Emulate the programming techniques that the C++ features were designed to support with the facilities provided by C.
- When writing C, write in the C subset of C++.

- Use compiler warning levels that ensure function argument checking.
- Use lint for large programs (see §27.2.2).

Many of the details of C/C++ incompatibilities are rather obscure and technical. However, to read and write C, you don't actually have to remember most of those:

- The compiler will remind you when you are using a C++ feature that is not in C.
- If you follow the rules above, you are unlikely to encounter anything that means something different in C from what it means in C++.

With the absence of all those C++ facilities, some facilities gain importance in C:

- Arrays and pointers
- Macros
- **typedef** (the C and C++98 equivalent to simple using declarations; see §20.5, §A.16)
- **sizeof**
- Casts

We give examples of a few such uses in this chapter.

I introduced the `//` comments into C++ from C's ancestor BCPL when I got really fed up with typing `/* ... */` comments. The `//` comments are accepted by most C dialects including C99 and C11, so it is probably safe just to use them. Here, we will use `/* ... */` exclusively in examples meant to be C. C99 and C11 introduced a few more C++ features (as well as a few features that are incompatible with C++), but here we will stick to C89, because that's far more widely used.



### 27.1.3 The C standard library

Naturally, a C++ library facility that depends on classes and templates is not available in C. This includes

- **vector**
- **map**
- **set**
- **string**
- The STL algorithms: e.g., **sort()**, **find()**, and **copy()**
- **iostreams**
- **regex**



For these, there are often C libraries based on arrays, pointers, and functions to help compensate. The main parts of the C standard library are

- `<stdlib.h>`: general utilities (e.g., `malloc()` and `free()`; see §27.4)
- `<stdio.h>`: standard I/O; see §27.6
- `<string.h>`: C-style string manipulation and memory manipulation; see §27.5
- `<math.h>`: standard floating-point mathematical functions; see §24.8
- `<errno.h>`: error codes for `<math.h>`; see §24.8
- `<limits.h>`: sizes of integer types; see §24.2
- `<time.h>`: date and time; see §26.6.1
- `<assert.h>`: debug assertions; see §27.9
- `<ctype.h>`: character classification; see §11.6
- `<stdbool.h>`: Boolean macros

For a complete description, see a good C textbook, such as K&R. All of these libraries (and header files) are also available in C++.

## 27.2 Functions

In C:

- There can be only one function of a given name.
- Function argument type checking is optional.
- There are no references (and therefore no pass-by-reference).
- There are no member functions.
- There are no inline functions (except in C99).
- There is an alternative function definition syntax.

Apart from that, things are much as you are used to in C++. Let us explore what that means.

### 27.2.1 No function name overloading

Consider:

```
void print(int);           /* print an int */  
void print(const char*);   /* print a string */ /* error! */
```

The second declaration is an error because there cannot be two functions with the same name. So you'll have to invent a suitable pair of names:

```
void print_int(int);           /* print an int */
void print_string(const char*); /* print a string */
```

This is occasionally claimed to be a virtue: now you can't accidentally use the wrong function to print an `int`! Clearly we don't buy that argument, and the lack of overloaded functions does make generic programming ideas awkward to implement because generic programming depends on semantically similar functions having the same name.

### 27.2.2 Function argument type checking

Consider:

```
int main()
{
    f(2);
}
```

A C compiler will accept this: you don't have to declare a function before you call it (though you can and should). There may be a definition of `f()` somewhere. That `f()` could be in another translation unit, but if it isn't, the linker will complain.

Unfortunately, that definition in another source file might look like this:

```
/* other_file.c: */

int f(char* p)
{
    int r = 0;
    while (*p++) r++;
    return r;
}
```

The linker will not report that error. You will get a run-time error or some random result.

How do we manage problems like that? Consistent use of header files is a practical answer. If every function you call or define is declared in a header that is consistently `#included` whenever needed, we get checking. However, in large programs that can be hard to achieve. Consequently, most C compilers have options that give warnings for calls of undeclared functions: use them. Also, from the earliest days of C, there have been programs that can be used to check for all kinds of consistency problems. They are usually called *lint*. Use a *lint* for every nontrivial C program. You will find that *lint* pushes you toward a style of C usage that is rather similar to using a subset of C++. One of the observations that led

to the design of C++ was that the compiler could easily check much (but not all) of what lint checked.

You can ask to have function arguments checked in C. You do that simply by declaring a function with its argument types specified (just as in C++). Such a declaration is called a *function prototype*. However, beware of function declarations that do not specify arguments; those are *not* function prototypes and do not imply function argument checking:

```
int g(double);      /* prototype — like C++ function declaration */
int h();           /* not a prototype — the argument types are unspecified */

void my_fct()
{
    g();           /* error: missing argument */
    g("asdf");     /* error: bad argument type */
    g(2);          /* OK: 2 is converted to 2.0 */
    g(2,3);        /* error: one argument too many */

    h();           /* OK by the compiler! May give unexpected results */
    h("asdf");     /* OK by the compiler! May give unexpected results */
    h(2);          /* OK by the compiler! May give unexpected results */
    h(2,3);        /* OK by the compiler! May give unexpected results */
}
```

The declaration of **h()** specifies no argument type. This does not mean that **h()** doesn't accept arguments; it means "Accept any set of arguments and hope they are correct for the called function." Again, a good compiler warns and lint will catch the problem.

C++	C equivalent
<b>void f(); // preferred</b>	<b>void f(void);</b>
<b>void f(void);</b>	<b>void f(void);</b>
<b>void f(. . .); // accept any arguments</b>	<b>void f(); /* accept any arguments */</b>

There is a special set of rules for converting arguments where no function prototype is in scope. For example, **chars** and **shorts** are converted to **ints**, and **floats** are converted to **doubles**. If you need to know, say, what happens to a **long**, look it up in a good C textbook. Our recommendation is simple: don't call functions without prototypes.

Note that even though the compiler will allow an argument of the wrong type to be passed, such as a `char*` to a parameter of type `int`, the use of such an argument of a wrong type is an error. As Dennis Ritchie said, “C is a strongly typed, weakly checked, programming language.”

### 27.2.3 Function definitions

You can define functions exactly as in C++ and such definitions are function prototypes:

```
double square(double d)
{
    return d*d;
}

void ff()
{
    double x = square(2);          /* OK: convert 2 to 2.0 and call */
    double y = square();           /* argument missing */
    double y = square("Hello");   /* error: wrong argument type */
    double y = square(2,3);       /* error: too many arguments */
}
```

A definition of a function with no arguments is not a function prototype:

```
void f() /* do something */

void g()
{
    f(2);      /* OK in C; error in C++ */
}
```

Having

```
void f(); /* no argument type specified */
```

mean “`f()` can take any number of arguments of any type” seemed really strange. In response, I invented a new notation where “nothing” was explicitly stated using the keyword `void` (`void` is a four-letter word meaning “nothing”):

```
void f(void); /* no arguments accepted */
```

I soon regretted that, though, since that looks odd and is completely redundant when argument type checking is uniformly applied. Worse, Dennis Ritchie (the father of C) and Doug McIlroy (the ultimate arbiter of taste in the Bell Labs Computer Science Research Center; see §22.2.5) both called it “an abomination.” Unfortunately, that abomination became very popular in the C community. Don’t use it in C++, though, where it is not only ugly, but also logically redundant.

C also provides a second, Algol60-style function definition, where the parameter types are (optionally) specified separately from their names:

```
int old_style(p,b,x) char* p; char b;  
{  
    /* . . . */  
}
```

This “old-style definition” predates C++ and is not a prototype. By default, an argument without a declared type is an `int`. So, `x` is an `int` parameter of `old_style()`. We can call `old_style()` like this:

```
old_style();           /* OK: all arguments missing */  
old_style("hello", 'a', 17); /* OK: all arguments are of the right type */  
old_style(12, 13, 14);   /* OK: 12 is the wrong type, */  
                      /* but maybe old_style() won't use p */
```

The compiler should accept these calls (but would warn, we hope, for the first and third).

Our recommendation about function argument checking:

- Use function prototypes consistently (use header files).
- Set compiler warning levels so that argument type errors are caught.
- Use (some) lint.

The result will be code that’s also C++.

#### 27.2.4 Calling C from C++ and C++ from C

You can link files compiled with a C compiler together with files compiled with a C++ compiler provided the two compilers were designed for that. For example, you can link object files generated from C and C++ using your GNU C and C++ compiler (GCC) together. You can also link object files generated from C and C++ using your Microsoft C and C++ compiler (MSC++) together. This is common and useful because it allows you to use a larger set of libraries than would be available in just one of those two languages.

C++ provides stricter type checking than C. In particular, a C++ compiler and linker check that two functions `f(int)` and `f(double)` are consistently defined

and used – even in different source files. A linker for C doesn't do that kind of checking. To call a function defined in C from C++ and to have a function defined in C++ called from C, we need to tell the compiler what we are doing:

```
// calling C function from C++:

extern "C" double sqrt(double);      // link as a C function

void my_c_plus_plus_fct()
{
    double sr = sqrt(2);
}
```

Basically **extern "C"** tells the compiler to use C linker conventions. Apart from that, all is normal from a C++ point of view. In fact, the C++ standard **sqrt(double)** usually is the C standard library **sqrt(double)**. Nothing is required from the C program to make a function callable from C++ in this way. C++ simply adapts to the C linkage convention.

We can also use **extern "C"** to make a C++ function callable from C:

```
// C+ function callable from C:

extern "C" int call_f(S* p, int i)
{
    return p->f(i);
}
```

In a C program, we can now call the member function **f()** indirectly, like this:

```
/* call C+ function from C: */

int call_f(S* p, int i);
struct S* make_S(int,const char*);

void my_c_fct(int i)
{
    /* ... */
    struct S* p = make_S(x, "foo");
    int x = call_f(p,i);
    /* ... */
}
```

No mention of C++ is needed (or possible) in C for this to work.

The benefit of this interoperability is obvious: code can be written in a mix of C and C++. In particular, a C++ program can use libraries written in C, and C programs can use libraries written in C++. Furthermore, most languages (notably Fortran) have an interface for calling to/from C.

In the examples above, we assumed that C and C++ could share the class object pointed to by **p**. That is true for most class objects. In particular, if you have a class like this,

```
// in C++:  
class complex {  
    double re, im;  
public:  
    // all the usual operations  
};
```

you can get away with passing a pointer to an object to and from C. You can even access **re** and **im** in a C program using a declaration:

```
/* in C: */  
struct complex {  
    double re, im;  
    /* no operations */  
};
```

The rules for layout in any language can be complex, and the rules for layout among languages can even be hard to specify. However, you can pass built-in types between C and C++ and also classes (**structs**) without virtual functions. If a class has virtual functions, you should just pass pointers to its objects and leave the actual manipulation to C++ code. The **call\_f()** was an example of this: **f()** might be **virtual** and then that example would illustrate how to call a virtual function from C.

Apart from sticking to the built-in types, the simplest and safest sharing of types is a **struct** defined in a common C/C++ header file. However, that strategy seriously limits how C++ can be used, so we don't restrict ourselves to it.

### 27.2.5 Pointers to functions

What can we do in C if we want to use object-oriented techniques (§14.2–4)? Basically, we need an alternative to virtual functions. For most people, the first idea that springs to mind is to use a **struct** with a “type field” that describes what kind of shape a given object represents. For example:

```

struct Shape1 {
    enum Kind { circle, rectangle } kind;
    /* . . . */
};

void draw(struct Shape1* p)
{
    switch (p->kind) {
        case circle:
            /* draw as circle */
            break;
        case rectangle:
            /* draw as rectangle */
            break;
    }
}

int f(struct Shape1* pp)
{
    draw(pp);
    /* . . . */
}

```

This works. There are two snags, though:

- For each “pseudo-virtual” function (such as `draw()`), we have to write a new `switch`-statement.
- Each time we add a new shape, we have to modify every “pseudo-virtual” function (such as `draw()`) by adding a case to the `switch`-statement.

The second problem is quite nasty because it means that we can’t provide our “pseudo-virtual” functions as part of a library, because our users will have to modify those functions quite often. The most effective alternative involves pointers to functions:

```

typedef void (*Pfct0)(struct Shape2*);
typedef void (*Pfct1int)(struct Shape2*,int);

struct Shape2 {
    Pfct0 draw;
    Pfct1int rotate;
    /* . . . */
};

```

```

void draw(struct Shape2* p)
{
    (p->draw)(p);
}

void rotate(struct Shape2* p, int d)
{
    (p->rotate)(p,d);
}

```

This **Shape2** can be used just like **Shape1**.

```

int f(struct Shape2* pp)
{
    draw(pp);
    /* . . . */
}

```

With a little extra work, an object need not hold one pointer to a function for each pseudo-virtual function. Instead, it can hold a pointer to an array of pointers to functions (much as virtual functions are implemented in C++). The main problem with using such schemes in real-world programs is to get the initialization of all those pointers to functions right.

## 27.3 Minor language differences

This section gives examples of minor C/C++ differences that could trip you up if you have never heard of them. Few seriously impact programming in that the differences have obvious work-arounds.

### 27.3.1 struct tag namespace

In C, the names of **structs** (there is no **class** keyword) are in a separate namespace from other identifiers. Therefore, every name of a **struct** (called a *structure tag*) must be prefixed with the keyword **struct**. For example:

```

struct pair { int x,y; };
pair p1;           /* error: no identifier pair in scope */
struct pair p2;     /* OK */
int pair = 7;       /* OK: the struct tag pair is not in scope */
struct pair p3;     /* OK: the struct tag pair is not hidden by the int */
pair = 8;          /* OK: pair refers to the int */

```

Amazingly enough, thanks to a devious compatibility hack, this also works in C++. Having a variable (or a function) with the same name as a `struct` is a fairly common C idiom, though not one we recommend.

If you don't want to write `struct` in front of every structure name, use a  `typedef` (§20.5). The following idiom is common:

```
typedef struct { int x,y; } pair;
pair p1 = { 1, 2 };
```

In general, you'll find `typedefs` more common and more useful in C, where you don't have the option of defining new types with associated operations.

In C, names of nested `structs` are placed in the same scope as the `struct` in which they are nested. For example: 

```
struct S {
    struct T /* ... */;
/* ... */
};

struct T x; /* OK in C (not in C++) */
```

In C++, you would write

```
S::T x; // OK in C++ (not in C)
```

Whenever possible, don't nest `structs` in C: their scope rules differ from what most people naively (and reasonably) expect.

### 27.3.2 Keywords

Many keywords in C++ are not keywords in C (because C doesn't provide the functionality) and can be used as identifiers in C:

C++ keywords that are not C keywords				
<code>alignas</code>	<code>class</code>	<code>inline</code>	<code>private</code>	<code>true</code>
<code>alignof</code>	<code>compl</code>	<code>mutable</code>	<code>protected</code>	<code>try</code>
<code>and</code>	<code>concept</code>	<code>namespace</code>	<code>public</code>	<code>typeid</code>
<code>and_eq</code>	<code>const_cast</code>	<code>new</code>	<code>reinterpret_cast</code>	<code>typename</code>
<code>asm</code>	<code>constexpr</code>	<code>noexcept</code>	<code>requires</code>	<code>using</code>
<code>bitand</code>	<code>delete</code>	<code>not</code>	<code>static_assert</code>	<code>virtual</code>

### C++ keywords that are not C keywords (*continued*)

<b>bitor</b>	<b>dynamic_cast</b>	<b>not_eq</b>	<b>static_cast</b>	<b>wchar_t</b>
<b>bool</b>	<b>explicit</b>	<b>nullptr</b>	<b>template</b>	<b>xor</b>
<b>catch</b>	<b>export</b>	<b>operator</b>	<b>this</b>	<b>xor_eq</b>
<b>char16_t</b>	<b>false</b>	<b>or</b>	<b>thread_local</b>	
<b>char32_t</b>	<b>friend</b>	<b>or_eq</b>	<b>throw</b>	



Don't use these names as identifiers in C, or your code will not be portable to C++. If you use one of these names in a header file, that header won't be useful from C++.

Some C++ keywords are macros in C:

### C++ keywords that are C macros

<b>and</b>	<b>bitor</b>	<b>false</b>	<b>or</b>	<b>wchar_t</b>
<b>and_eq</b>	<b>bool</b>	<b>not</b>	<b>or_eq</b>	<b>xor</b>
<b>bitand</b>	<b>compl</b>	<b>not_eq</b>	<b>true</b>	<b>xor_eq</b>

In C, they are defined in `<iso646.h>` and `<stdbool.h>` (`bool`, `true`, `false`). Don't take advantage of the fact that they are macros in C.

### 27.3.3 Definitions

C++ allows definitions in more places than C89. For example:

```

for (int i = 0; i<max; ++i) x[i] = y[i];      // definition of i not allowed in C

while (struct S* p = next(q)) {           // definition of p not allowed in C
    /* . . . */
}

void f(int i)
{
    if (i< 0 || max<=i) error("range error");
    int a[max];      // error: declaration after statement not allowed in C
    /* . . . */
}

```

C (C89) doesn't allow declarations as initializers in `for`-statements, as conditions, or after a statement in a block. We have to write something like

```

int i;
for (i = 0; i<max; ++i) x[i] = y[i];

struct S* p;
while (p = next(q)) {
    /* . . . */
}

void f(int i)
{
    if (i< 0 || max<=i) error("range error");
    {
        int a[max];
        /* . . . */
    }
}

```

In C++, an uninitialized declaration is a definition; in C, it is just a declaration so that there can be two of them:

```

int x;
int x;      /* defines or declares a single integer called x in C; error in C++ */

```

In C++, an entity must be defined exactly once. This gets a bit more interesting if the two **ints** are in different translation units:

```

/* in file x.c: */
int x;

/* in file y.c: */
int x;

```

No C or C++ compiler will find any fault with either **x.c** or **y.c**. However, if **x.c** and **y.c** are compiled as C++, the linker will give a “double definition” error. If **x.c** and **y.c** are compiled as C, the linker accepts the program and (correctly according to C rules) considers there to be just one **x** that is shared between code in **x.c** and **y.c**. If you want a program where a global variable **x** is shared, say so explicitly:

```

/* in file x.c: */
int x = 0;          /* the definition */

/* in file y.c: */
extern int x;       /* a declaration, not a definition */

```

Better still, use a header file:

```
/* in file x.h: */
extern int x;           /* a declaration, not a definition */

/* in file x.c: */
#include "x.h"
int x = 0;            /* the definition */

/* in file y.c: */
#include "x.h"
/* the declaration of x is in the header */
```

Better still, avoid the global variable.

### 27.3.4 C-style casts

In C (and C++), you can explicitly convert a value **v** to a type **T** by this minimal notation:

**(T)v**

This “C-style cast” or “old-style cast” is beloved by poor typists and sloppy thinkers because it’s minimal and you don’t have to know what it takes to make a **T** from **v**. On the other hand, this style of cast is rightfully feared by maintenance programmers because it is just about invisible and leaves no clue about the writer’s intent. The C++ casts (*named casts* or *template-style casts*; see §A.5.7) were introduced to make explicit type conversion easy to spot (ugly) and specific. In C, you have no choice:

```
int* p = (int*)7;      /* reinterpret bit pattern: reinterpret_cast<int*>(7) */
int x = (int)7.5;       /* truncate double: static_cast<int>(7.5) */

typedef struct S1 { /* ... */ } S1;
typedef struct S2 { /* ... */ } S2;
S2 a;
const S2 b;            /* uninitialized consts are allowed in C */

S1* p = (S1*)&a;     /* reinterpret bit pattern: reinterpret_cast<S1*>(&a) */
S2* q = (S2*)&b;     /* cast away const: const_cast<S2*>(&b) */
S1* r = (S1*)&b;     /* remove const and change type; probably a bug */
```

We hesitate to recommend a macro (§27.8) even in C, but it may be an idea to express intent like this:

```
#define REINTERPRET_CAST(T,v) ((T)(v))
#define CONST_CAST(T,v) ((T)(v))

S1* p = REINTERPRET_CAST (S1*,&a);
S2* q = CONST_CAST(S2*,&b);
```

This does not give the type checking done by `reinterpret_cast` and `const_cast`, but it does make these inherently ugly operations visible and the programmer's intent explicit.

### 27.3.5 Conversion of `void*`

In C, a `void*` may be used as the right-hand operand of an assignment to or initialization of a variable of any pointer type; in C++ it may not. For example:

```
void* alloc(size_t x);           /* allocate x bytes */

void f (int n)
{
    int* p = alloc(n*sizeof(int)); /* OK in C; error in C++ */
    /* . . . */
}
```

Here, the `void*` result of `alloc()` is implicitly converted to an `int*`. In C++, we would have to rewrite that line to

```
int* p = (int*)alloc(n*sizeof(int)); /* OK in C and C++ */
```

We used the C-style cast (§27.3.4) so that it would be legal in both C and C++.

Why is the `void*-to-T*` implicit conversion illegal in C++? Because such conversions can be unsafe:

```
void f()
{
    char i = 0;
    char j = 0;
    char* p = &i;
    void* q = p;
    int* pp = q;          /* unsafe; legal in C, error in C++ */
    *pp = -1;             /* overwrite memory starting at &i */
}
```



Here we can't even be sure what memory is overwritten. Maybe **j** and part of **p**? Maybe some memory used to manage the call of **f()** (**f**'s stack frame)? Whatever data is being overwritten here, a call of **f()** is bad news.

Note that (the opposite) conversion of a **T\*** to a **void\*** is perfectly safe – you can't construct nasty examples like the one above for that – and those are allowed in both C and C++.

Unfortunately, implicit **void\***-to-**T\*** conversions are common in C and possibly the major C/C++ compatibility problem in real code (see §27.4).

27.3.6 enum

In C, you can assign an **int** to an **enum** without a cast. For example:

```
enum color { red, blue, green };
int x = green;           /* OK in C and C++ */
enum color col = 7;     /* OK in C; error in C++ */
```

One implication of this is that we can use increment (`++`) and decrement (`--`) on variables of enumeration type in C. That can be convenient but does imply a hazard:

```
enum color x = blue;  
++x;      /* x becomes green; error in C++ */  
++x;      /* x becomes 3; error in C++ */
```

“Falling off the end” of the enumerators may or may not have been what we wanted.

Note that like structure tags, the names of enumerations are in their own namespace, so you have to prefix them with the keyword **enum** each time you use them:

```
color c2 = blue;           /* error in C: color not in scope; OK in C++ */  
enum color c3 = red;       /* OK */
```

### 27.3.7 Namespaces

There are no namespaces (in the C++ sense of the word) in C. So what do you do when you want to avoid name clashes in large C programs? Typically, people use prefixes or suffixes. For example:

```
/* in pete.h: */
typedef char* pete_string;           /* Pete's string */
typedef char pete_bool;            /* Pete's Boolean type */
```

This technique is so popular that it is usually a bad idea to use one- or two-letter prefixes.

## 27.4 Free store

C does not provide the **new** and **delete** operators dealing with objects. To use the free store, you use functions dealing with memory. The most important functions are defined in the “general utilities” standard header **<stdlib.h>**:

```
void* malloc(size_t sz);           /* allocate sz bytes */
void free(void* p);             /* deallocate the memory pointed to by p */
void* calloc(size_t n, size_t sz); /* allocate n*sz bytes initialized to 0 */
void* realloc(void* p, size_t sz); /* reallocate the memory pointed to by p
to a space of size sz */
```

The **typedef size\_t** is an unsigned type also defined in **<stdlib.h>**.

Why does **malloc()** return a **void\***? Because **malloc()** has no idea which type of object you want to put in that memory. Initialization is your problem. For example:

```
struct Pair {
    const char* p;
    int val;
};

struct Pair p2 = {"apple", 78};
struct Pair* pp = (struct Pair*) malloc(sizeof(Pair));           /* allocate */
pp->p = "pear";           /* initialize */
pp->val = 42;
```

Note that we cannot write

```
*pp = {"pear", 42};      /* error: not C or C++98 */
```

in either C or C++. However, in C++, we would define a constructor for **Pair** and write

```
Pair* pp = new Pair("pear", 42);
```

In C (but not C++; see §27.3.4), you can leave out the cast before **malloc()**, but we don't recommend that:

```
int* p = malloc(sizeof(int)*n); /* avoid this */
```

Leaving out the cast is quite popular because it saves some typing and because it catches the rare error of (illegally) forgetting to include **<stdlib.h>** before using **malloc()**. However, it can also remove a visual clue that a size was wrongly calculated:

```
p = malloc(sizeof(char)*m); /* probably a bug — not room for m ints */
```



Don't use **malloc()/free()** in C++ programs; **new/delete** require no casts, deal with initialization (constructors) and cleanup (destructors), report memory allocation errors (through an exception), and are just as fast. Don't **delete** an object allocated by **malloc()** or **free()** an object allocated by **new**. For example:

```
int* p = new int[200];
// ...
free(p); // error

X* q = (X*)malloc(n*sizeof(X));
// ...
delete q; // error
```

This might work, but it is not portable code. Furthermore, for objects with constructors or destructors, mixing C-style and C++-style free-store management is a recipe for disaster.

The **realloc()** function is typically used for expanding buffers:

```
int max = 1000;
int count = 0;
int c;
char* p = (char*)malloc(max);
while ((c=getchar())!=EOF) { /* read: ignore chars on eof line */
    if (count==max-1) { /* need to expand buffer */
        max += max; /* double the buffer size */
        p = (char*)realloc(p,max);
        if (p==0) quit();
    }
    p[count++] = c;
}
```

For an explanation of the C input operations, see §27.6.2 and §B.11.2.

The `realloc()` function may or may not move the old allocation into newly allocated memory. Don't even think of using `realloc()` on memory allocated by `new`.

Using the C++ standard library, the (roughly) equivalent code is

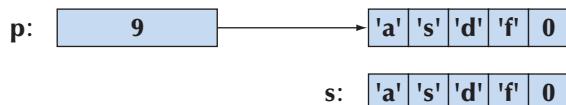
```
vector<char> buf;
char c;
while (cin.get(c)) buf.push_back(c);
```

Refer to the paper “Learning Standard C++ as a New Language” (see the reference list in §27.1) for a more thorough discussion of input and allocation strategies.

## 27.5 C-style strings

In C, a string (often called a *C string* or a *C-style string* in C++ literature) is a zero-terminated array of characters. For example:

```
char* p = "asdf";
char s[] = "asdf";
```



In C, we cannot have member functions, we cannot overload functions, and we cannot define an operator (such as `==`) for a `struct`. It follows that we need a set of (nonmember) functions to manipulate C-style strings. The C and C++ standard libraries provide such functions in `<string.h>`:

```
size_t strlen(const char* s);           /* count the characters */
char* strcat(char* s1, const char* s2);  /* copy s2 onto the end of s1 */
int strcmp(const char* s1, const char* s2); /* compare lexicographically */
char* strcpy(char* s1, const char* s2);   /* copy s2 into s1 */

char* strchr(const char *s, int c);      /* find c in s */
char* strstr(const char *s1, const char *s2); /* find s2 in s1 */

char* strncpy(char*, const char*, size_t n); /* strcpy, max n chars */
char* strncat(char*, const char, size_t n);  /* strcat with max n chars */
int strncmp(const char*, const char*, size_t n); /* strcmp with max n chars */
```

This is not the full set, but these are the most useful and most used functions. We will briefly illustrate their use.

We can compare strings. The equality operator (`==`) compares pointer values; the standard library function `strcmp()` compares C-style string values:

```
const char* s1 = "asdf";
const char* s2 = "asdf";

if (s1==s2) { /* do s1 and s2 point to the same array? */
    /* (typically not what you want) */
}

if (strcmp(s1,s2)==0) { /* do s1 and s2 hold the same characters? */
}
```

The `strcmp()` function does a three-way comparison of its two arguments. Given the values of `s1` and `s2` above, `strcmp(s1,s2)` will return `0`, meaning a perfect match. If `s1` was lexicographically before `s2`, it would return a negative number, and if `s1` was lexicographically after `s2`, it would return a positive number. The term *lexicographical* means roughly “as in a dictionary.” For example:

```
strcmp("dog","dog")==0
strcmp("ape","dodo")<0 /* "ape" comes before "dodo" in a dictionary */
strcmp("pig","cow")>0 /* "pig" comes after "cow" in a dictionary */
```

The value of the pointer comparison `s1==s2` is not guaranteed to be `0` (`false`). An implementation may decide to use the same memory to hold all copies of a character literal, so we would get the answer `1` (`true`). Usually, `strcmp()` is the right choice for comparing C-style strings.

We can find the length of a C-style string using `strlen()`:

```
int lgt = strlen(s1);
```

Note that `strlen()` counts characters excluding the terminating 0. In this case, `strlen(s1)==4` and it takes 5 bytes to store `"asdf"`. This little difference is the source of many off-by-one errors.

We can copy one C-style string (including the terminating 0) into another:

```
strcpy(s1,s2); /* copy characters from s2 into s1 */
```

It is your job to be sure that the target string (array) has enough space to hold the characters from the source.

The **strncpy()**, **strncat()**, and **strcmp()** functions are versions of **strcpy()**, **strcat()**, and **strcmp()** that will consider a maximum of **n** characters, where **n** is their third argument. Note that if there are more than **n** characters in the source string, **strncpy()** will not copy a terminating 0, so that the result will not be a valid C-style string.

The **strchr()** and **strstr()** functions find their second argument in the string that is their first argument and return a pointer to the first character of the match. Like **find()**, they search from left to right in the string.

It is amazing both how much can be done with these simple functions and how easy it is to make minor mistakes. Consider a simple problem of concatenating a user name with an address, placing the **@** character in between. Using **std::string** this can be done like this:

```
string s = id + '@' + addr;
```

Using the standard C-style string function we can write that as

```
char* cat(const char* id, const char* addr)
{
    int sz = strlen(id)+strlen(addr)+2;
    char* res = (char*) malloc(sz);
    strcpy(res,id);
    res[strlen(id)+1] = '@';
    strcpy(res+strlen(id)+2,addr);
    res[sz-1]=0;
    return res;
}
```

Did we get that right? Who will **free()** the string returned from **cat()**?

### TRY THIS



Test **cat()**. Why **2**? We left a beginner's performance error in **cat()**; find it and remove it. We "forgot" to comment our code. Add comments suitable for someone who can be assumed to know the standard C-string functions.

#### 27.5.1 C-style strings and **const**

Consider:

```
char* p = "asdf";
p[2] = 'x';
```



This is legal in C but not in C++. In C++, a string literal is a constant, an immutable value, so `p[2]='x'` (to make the value pointed to "asxf") is illegal. Unfortunately, few compilers will catch the assignment to `p` that leads to the problem. If you are lucky, a run-time error will occur, but don't rely on that. Instead, write

```
const char* p = "asdf"; // now you can't write to "asdf" through p
```

This recommendation applies to both C and C++.

The C `strchr()` has a similar but even harder-to-spot problem. Consider:



```
char* strchr(const char* s, int c); /* find c in constant s (not C++) */

const char aa[] = "asdf";           /* aa is an array of constants */
char* q = strchr(aa, 'd');        /* finds 'd' */
*q = 'x';                         /* change 'd' in aa to 'x' */
```

Again, this is illegal in C and C++, but C compilers can't catch it. Sometimes this is referred to as *transmutation*: it turns `consts` into non-`consts`, violating reasonable assumptions about code.

In C++, the problem is solved by the standard library declaring `strchr()` differently:

```
char const* strchr(const char* s, int c); // find c in constant s
char* strchr(char* s, int c);             // find c in s
```

Similarly for `strstr()`.

### 27.5.2 Byte operations

In the distant dark ages (the early 1980s), before the invention of `void*`, C (and C++) programmers used the string operations to manipulate bytes. Now the basic memory manipulation standard library functions have `void*` parameters and return types to warn users about their direct manipulation of essentially untyped memory:

```
/* copy n bytes from s2 to s1 (like strcpy): */
void* memcpy(void* s1, const void* s2, size_t n);

/* copy n bytes from s2 to s1 ( [s1:s1+n) may overlap with [s2:s2+n) ): */
void* memmove(void* s1, const void* s2, size_t n);

/* compare n bytes from s2 to s1 (like strcmp): */
int memcmp(const void* s1, const void* s2, size_t n);
```

```
/* find c (converted to an unsigned char) in the first n bytes of s: */
void* memchr(const void* s, int c, size_t n);

/* copy c (converted to an unsigned char)
   into each of the first n bytes that s points to: */
void* memset(void* s, int c, size_t n);
```

Don't use these functions in C++. In particular, **memset()** typically interferes with the guarantees offered by constructors.

### 27.5.3 An example: **strcpy()**

The definition of **strcpy()** is both famous and infamous as an example of the terse style that C (and C++) is capable of:

```
char* strcpy(char* p, const char* q)
{
    while (*p++ = *q++);
    return p;
}
```

We leave to you the explanation of why this actually copies the C-style string **q** into **p**. Post-increment is described in §A.5: The value of **p++** is the value of **p** before increment.

---

#### TRY THIS



Is this implementation of **strcpy()** correct? Explain why.

If you can't explain why, we won't consider you a C programmer (however competent you are at programming in other languages). Every language has its own idioms, and this is one of C's.



### 27.5.4 A style issue

We have quietly taken sides in a long-standing, often furiously debated, and largely irrelevant style issue. We declare a pointer like this:

**char\* p;**      *// p is a pointer to a char*

and not like this:

**char \*p;**      */\* p is something that you can dereference to get a char \*/*

The placement of the whitespace is completely irrelevant to the compiler, but programmers care. Our style (common in C++) emphasizes the type of the variable being declared, whereas the other style (more common in C) emphasizes the use of the variable. Note that we don't recommend declaring many variables in a single declaration:

```
char c, *p, a[177], *f(); /* legal, but confusing */
```

Such declarations are not uncommon in older code. Instead, use multiple lines and take advantage of the extra horizontal space for comments and initializers:

```
char c = 'a';           /* termination character for input using f() */
char* p = 0;            /* last char read by f() */
char a[177];           /* input buffer */
char* f();              /* read into buffer a; return pointer to first char read */
```

Also, choose meaningful names.

## 27.6 Input/output: stdio

There are no **iostreams** in C, so we use the C standard I/O defined in **<stdio.h>** and commonly referred to as stdio. The stdio equivalents to **cin** and **cout** are **stdin** and **stdout**. Stdio and **iostream** use can be mixed in a single program (for the same I/O streams), but we don't recommend that. If you feel the need to mix, read up on stdio and **iostreams** (especially **ios\_base::sync\_with\_stdio()**) in an expert-level textbook. See also §B.11.

### 27.6.1 Output

The most popular and useful function of stdio is **printf()**. The most basic use of **printf()** just prints a (C-style) string:

```
#include<stdio.h>

void f(const char* p)
{
    printf("Hello, World!\n");
    printf(p);
}
```

That's not particularly interesting. The interesting bit is that `printf()` can take an arbitrary number of arguments, and the initial string controls if and how those extra arguments are printed. The declaration of `printf()` in C looks like this:

```
int printf(const char* format, ...);
```

The `...` means “and optionally more arguments.” We can call `printf()` like this:

```
void f1(double d, char* s, int i, char ch)
{
    printf("double %g string %s int %d char %c\n", d, s, i, ch);
}
```

Here, `%g` means “Print a floating-point number using the general format,” `%s` means “Print a C-style string,” `%d` means “Print an integer using decimal digits,” and `%c` means “Print a character.” Each such format specifier picks the next so-far-unused argument, so `%g` prints `d`, `%s` prints `s`, `%d` prints `i`, and `%c` prints `ch`. You can find the full list of `printf()` formats in §B.11.2.

Unfortunately, `printf()` is not type safe. For example:

```
char a[] = { 'a', 'b' };           /* no terminating 0 */

void f2(char* s, int i)
{
    printf("goof %s\n", i);        /* uncaught error */
    printf("goof %d: %s\n", i);    /* uncaught error */
    printf("goof %s\n", a);        /* uncaught error */

}
```

The effect of the last `printf()` is interesting: it prints every byte in memory following `a[1]` until it encounters a 0. That could be a lot of characters.

This lack of type safety is one reason we prefer `iostreams` over stdio even though stdio works identically in C and C++. The other reason is that the stdio functions are not extensible: you cannot extend `printf()` to print values of your own types, the way you can using `iostreams`. For example, there is no way you can define your own `%Y` to print some `struct Y`.

There is a useful version of `printf()` that takes a file descriptor as its first argument:

```
int fprintf(FILE* stream, const char* format, ...);
```

For example:

```
fprintf(stdout,"Hello, World!\n"); // exactly like printf("Hello, World!\n");
FILE* ff = fopen("My_file", "w"); // open My_file for writing
fprintf(ff,"Hello, World!\n"); // write "Hello, World!\n" to My_file
```

File handles are described in §27.6.3.

## 27.6.2 Input

The most popular stdio functions include

```
int scanf(const char* format, . . .); /* read from stdin using a format */
int getchar(void); /* get a char from stdin */
int getc(FILE* stream); /* get a char from stream */
char* gets(char* s); /* get characters from stdin */
```

The simplest way of reading a string of characters is using **gets()**. For example:

```
char a[12];
gets(a); /* read into char array pointed to by a until a '\n' is input */
```

Never do that! Consider **gets()** poisoned. Together with its close cousin **scanf("%s")**, **gets()** used to be the root cause of about a quarter of all successful hacking attempts. It is still a major security problem. In the trivial example above, how would you know that at most 11 characters would be input before a newline? You can't know that. Thus, **gets()** almost certainly leads to memory corruption (of the bytes after the buffer), and memory corruption is a major tool of crackers. Don't think that you can guess a maximum buffer size that is "large enough for all uses." Maybe the "person" at the other end of the input stream is a program that does not meet your criteria for reasonableness.

The **scanf()** function reads using a format just as **printf()** writes using a format. Like **printf()** it can be very convenient:

```
void f()
{
    int i;
    char c;
    double d;
    char* s = (char*)malloc(100);
    /* read into variables passed as pointers: */
    scanf("%i %c %g %s", &i, &c, &d, s);
    /* %s skips initial whitespace and is terminated by whitespace */
}
```

Like `printf()`, `scanf()` is not type safe. The format characters and the arguments (all pointers) must match exactly, or strange things will happen at run time. Note also that the `%s` read into `s` may lead to an overflow. Don't ever use `gets()` or `scanf("%s")!`

So how do we read characters safely? We can use a form of `%s` that places a limit on the number of characters read. For example:

```
char buf[20];
scanf("%19s",buf);
```

We need space for a terminating 0 (supplied by `scanf()`), so 19 is the maximum number of characters we can read into `buf`. However, that leaves us with the problem of what to do if someone does type more than 19 characters. The “extra” characters will be left in the input stream to be “found” by later input operations.

The problem with `scanf()` implies that it is often prudent and easier to use `getchar()`. The typical way of reading characters with `getchar()` is

```
while((x=getchar())!=EOF) {
    /* . . . */
}
```

`EOF` is a stdio macro meaning “end of file”; see also §27.4.

The C++ standard library alternative to `scanf("%s")` and `gets()` doesn't suffer from these problems:

```
string s;
cin >> s;           // read a word
getline(cin,s);    // read a line
```

### 27.6.3 Files

In C (or C++), files can be opened using `fopen()` and closed using `fclose()`. These functions, together with the representation of a file handle, `FILE`, and the `EOF` (end-of-file) macro, are found in `<stdio.h>`:

```
FILE *fopen(const char* filename, const char* mode);
int fclose(FILE *stream);
```

Basically, you use files like this:

```
void f(const char* fn, const char* fn2)
{
    FILE* fi = fopen(fn, "r");          /* open fn for reading */
    FILE* fo = fopen(fn2, "w");         /* open fn2 for writing */
```

```

if (fi == 0) error("failed to open input file");
if (fo == 0) error("failed to open output file");

/* read from file using stdio input functions, e.g., getc()
/* write to file using stdio output functions, e.g., fprintf() */

fclose(fo);
fclose(fi);
}

```

Consider this: there are no exceptions in C, so how do we make sure that the files are closed whichever error happens?

## 27.7 Constants and macros

In C, a **const** is never a compile-time constant:

```

const int max = 30;
const int x;          /* const not initialized: OK in C (error in C++) */

void f(int v)
{
    int a1[max];   /* error: array bound not a constant (OK in C++) */
                    /* (max is not allowed in a constant expression!) */
    int a2[x];     /* error: array bound not a constant */

    switch (v) {
        case 1:
            /* . . . */
            break;
        case max:    /* error: case label not a constant (OK in C++) */
            /* . . . */
            break;
    }
}

```

The technical reason in C (though not in C++) is that a **const** is implicitly accessible from other translation units:

```

/* file x.c: */
const int x;          /* initialize elsewhere */

```

```
/* file xx.c: */
const int x = 7;      /* here is the real definition */
```

In C++, that would be two different objects, each called `x` in its own file. Instead of using `const` to represent symbolic constants, C programmers tend to use macros. For example:

```
#define MAX 30

void f(int v)
{
    int a1[MAX]; /* OK */

    switch (v) {
    case 1:
        /* ... */
        break;
    case MAX: /* OK */
        /* ... */
        break;
    }
}
```

The name of the macro `MAX` is replaced by the characters `30`, which is the value of the macro; that is, the number of elements of `a1` is `30` and the value in the second case label is `30`. We use all capital letters for the `MAX` macro, as is conventional. This naming convention helps minimize errors caused by macros.



## 27.8 Macros

Beware of macros: in C there are no really effective ways of avoiding macros, but their use has serious side effects because they don't obey the usual C (or C++) scope and type rules. Macros are a form of text substitution. See also §A.17.2.



How do we try to protect ourselves from the potential problems of macros apart from (relying on C++ alternatives and) minimizing their use?

- Give all macros we define `ALL_CAPS` names.
- Don't give anything that isn't a macro an `ALL_CAPS` name.
- Never give a macro a short or “cute” name, such as `max` or `min`.
- Hope that everybody else follows this simple and common convention.



The main uses of macros are

- Definition of “constants”
- Definition of function-like constructs
- “Improvements” to the syntax
- Control of conditional compilation

In addition, there is a wide variety of less common uses.

We consider macros seriously overused, but there are no reasonable and complete alternatives to the use of macros in C programs. It can even be hard to avoid them in C++ programs (especially if you need to write programs that have to be portable to very old compilers or to platforms with unusual constraints).

Apologies to people who consider the techniques described below “dirty tricks” and believe such are best not mentioned in polite company. However, we believe that programming is to be done in the real world and that these (very mild) examples of uses and misuses of macros can save hours of grief for the novice programmer. Ignorance about macros is not bliss.

### 27.8.1 Function-like macros

Here is a fairly typical function-like macro:

```
#define MAX(x, y) ((x)>=(y)?(x):(y))
```

We use the capital **MAX** to distinguish it from the many functions called **max** (in various programs). Obviously, this is very different from a function: there are no argument types, no block, no return statement, etc., and what are all those parentheses doing? Consider:

```
int aa = MAX(1,2);
double dd = MAX(aa++,2);
char cc = MAX(dd,aa)+2;
```

This expands to

```
int aa = ((1)>=( 2)?(1):(2));
double dd = ((aa++)>=(2)?( aa++):(2));
char cc = ((dd)>=(aa)?(dd):(aa))+2;
```

Had “all the parentheses” not been there, the last expansion would have ended up as

```
char cc = dd>=aa?dd:aa+2;
```

That is, `cc` could easily have gotten a different value from what you would reasonably expect looking at the definition of `cc`. When you define a macro, remember to put every use of an argument as an expression in parentheses.

On the other hand, not all the parentheses in the world could save the second expansion. The macro parameter `x` was given the value `aa++`, and since `x` is used twice in `MAX`, `a` can get incremented twice. Don't pass an argument with a side effect to a macro.

As it happens, some genius did define a macro like that and stuck it in a popular header file. Unfortunately, he also called it `max`, rather than `MAX`, so when the C++ standard header defines

```
template<class T> inline T max(T a,T b) { return a<b?b:a; }
```

the `max` gets expanded with the arguments `T a` and `T b`, and the compiler sees

```
template<class T> inline T ((T a)>=( T b)?( T a):( T b)) { return a<b?b:a; }
```

The compiler error messages are "interesting" and not very helpful. In an emergency, you can "undefine" a macro:

```
#undef max
```

Fortunately, that macro was not all that important. However, there are tens of thousands of macros in popular header files; you can't undefine them all without causing havoc.

Not all macro parameters are used as expressions. Consider:

```
#define ALLOC(T,n) ((T*)malloc(sizeof(T)*n))
```

This is a real example that can be very useful for avoiding errors stemming from a mismatch of the intended type of an allocation and its use in a `sizeof`:

```
double* p = malloc(sizeof(int)*10); /* likely error */
```

Unfortunately, it is nontrivial to write a macro that also catches memory exhaustion. This might do, provided that you define `error_var` and `error()` appropriately somewhere:

```
#define ALLOC(T,n) (error_var = (T*)malloc(sizeof(T)*n), \
                    (error_var==0)\ 
                     ?(error("memory allocation failure"),0)\ 
                     :error_var)
```

The lines ending with \ are not a typesetting problem; it is the way you break a macro definition across lines. When writing C++, we prefer to use **new**.

### 27.8.2 Syntax macros

You can define macros that make the source code look more to your taste. For example:

```
#define forever for(;;)
#define CASE break; case
#define begin {
#define end }
```

We strongly recommend against this. *Many* people have tried this idea. They (or the people who maintain their code) find that

- Many people don't share their idea of what is a better syntax.
- The "improved" syntax is nonstandard and surprising; others get confused.
- There are uses of the "improved" syntax that cause obscure compile-time errors.
- What you see is not what the compiler sees, and the compiler reports errors in the vocabulary it knows (and sees in source code), not in yours.

Don't write syntactic macros to "improve" the look of code. You and your best friends might find it really nice, but experience shows that you'll be a tiny minority in the larger community, so that someone will have to rewrite your code (assuming it survives).

### 27.8.3 Conditional compilation

Imagine you have two versions of a header file, say, one for Linux and one for Windows. How do you select in your code? Here is a common way:

```
#ifdef WINDOWS
    #include "my_windows_header.h"
#else
    #include "my_linux_header.h"
#endif
```

Now, if someone had defined **WINDOWS** before the compiler sees this, the effect is

```
#include "my_windows_header.h"
```

Otherwise it is

```
#include "my_linux_header.h"
```

The `#ifdef WINDOWS` test doesn't care what `WINDOWS` is defined to be; it just tests that it is defined.

Most major systems (including all operating system variants) have macros defined so that you can check. The check whether you are compiling as C++ or compiling as C is

```
#ifdef __cplusplus
    // in C++
#else
    /* in C */
#endif
```

A similar construct, often called an *include guard*, is commonly used to prevent a header file from being `#included` twice:

```
/* my_windows_header.h: */
#ifndef MY_WINDOWS_HEADER
#define MY_WINDOWS_HEADER
    /* here is the header information */
#endif
```

The `#ifndef` test checks that something is not defined; i.e., `#ifndef` is the opposite of `#ifdef`. Logically, these macros used for source file control are very different from the macros we use for modifying source code. They just happen to use the same underlying mechanisms to do their job.

## 27.9 An example: intrusive containers

The C++ standard library containers, such as `vector` and `map`, are non-intrusive; that is, they require no data in the types used as elements. That is how they generalize nicely to essentially all types (built-in or user-defined) as long as those types can be copied. There is another kind of container, an *intrusive container*, that is popular in both C and C++. We will use a non-intrusive list to illustrate C-style use of `structs`, pointers, and free store.

Let's define a doubly-linked list with nine operations:

```
void init(struct List* lst);           /* initialize lst to empty */
struct List* create();                 /* make a new empty list on free store */
void clear(struct List* lst);          /* free all elements of lst */
void destroy(struct List* lst);         /* free all elements of lst, then free lst */

void push_back(struct List* lst, struct Link* p); /* add p at end of lst */
void push_front(struct List*, struct Link* p);  /* add p at front of lst */
```

```

/* insert q before p in lst: */
void insert(struct List* lst, struct Link* p, struct Link* q);
struct Link* erase(struct List* lst, struct Link* p); /* remove p from lst */

/* return link n "hops" before or after p: */
struct Link* advance(struct Link* p, int n);

```

The idea is to define these operations so that their users need only use **List**\*'s and **Link**\*'s. This implies that the implementation of these functions could be changed radically without affecting those users. Obviously, the naming is influenced by the STL. **List** and **Link** can be defined in the obvious and trivial manner:

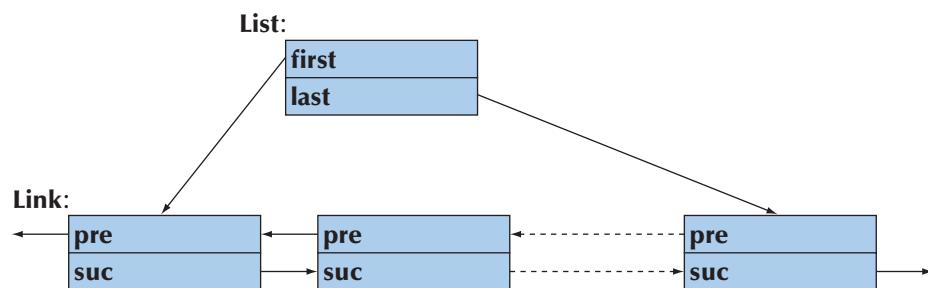
```

struct List {
    struct Link* first;
    struct Link* last;
};

struct Link { /* link for doubly-linked list */
    struct Link* pre;
    struct Link* suc;
};

```

Here is a graphical representation of a **List**:



It is not our aim to demonstrate clever representation techniques or clever algorithms, so there are none of those here. However, do note that there is no mention of any data held by the **Links** (the elements of a **List**). Looking back at the functions provided, we note that we are doing something very similar to defining a pair of abstract classes **Link** and **List**. The data for **Links** will be supplied later. **Link**\* and **List**\* are sometimes called handles to opaque types; that is, giving **Link**\*'s and **List**\*'s to our functions allows us to manipulate elements of a **List** without knowing anything about the internal structure of a **Link** or a **List**.

To implement our **List** functions, we first **#include** some standard library headers:

```
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>
```

C doesn't have namespaces, so we need not worry about **using** declarations or **using** directives. On the other hand, we should probably worry that we have grabbed some very common short names (**Link**, **insert**, **init**, etc.), so this set of functions cannot be used "as is" outside a toy program.

Initializing is trivial, but note the use of **assert()**:

```
void init(struct List* lst)      /* initialize *lst to the empty list */
{
    assert(lst);
    lst->first = lst->last = 0;
}
```

We decided not to deal with error handling for bad pointers to lists at run time. By using **assert()**, we simply give a (run-time) system error if a list pointer is null. The "system error" will give the file name and line number of the failed **assert()**; **assert()** is a macro defined in **<assert.h>** and the checking is enabled only during debugging. In the absence of exceptions, it is not easy to know what to do with bad pointers.

The **create()** function simply makes a **List** on the free store. It is a sort of combination of a constructor (**init()** initializes) and **new (malloc)** allocates):

```
struct List* create()          /* make a new empty list */
{
    struct List* lst = (struct List*)malloc(sizeof(struct List));
    init(lst);
    return lst;
}
```

The **clear()** function assumes that all **Links** are created on the free store and **free()** s them:

```
void clear(struct List* lst)    /* free all elements of lst */
{
    assert(lst);
{
```

```

struct Link* curr = lst->first;
while(curr) {
    struct Link* next = curr->suc;
    free(curr);
    curr = next;
}
lst->first = lst->last = 0;
}
}

```

Note the way we traverse using the **suc** member of **Link**. We can't safely access a member of a **struct** object after that object has been **free()**d, so we introduce the variable **next** to hold our position in the **List** while we **free()** a **Link**.

If we didn't allocate all of our **Links** on the free store, we had better not call **clear()**, or **clear()** will create havoc.

The **destroy()** function is essentially the opposite of **create()**, that is, a sort of combination of a destructor and a **delete**:

```

void destroy(struct List* lst) /* free all elements of lst; then free lst */
{
    assert(lst);
    clear(lst);
    free(lst);
}

```

Note that we are making no provisions for calling a cleanup function (destructor) for the elements represented by **Links**. This design is not a completely faithful imitation of C++ techniques or generality – it couldn't and probably shouldn't be.

The **push\_back()** function – adding a **Link** as the new last **Link** – is pretty straightforward:

```

void push_back(struct List* lst, struct Link* p) /* add p at end of lst */
{
    assert(lst);
    {
        struct Link* last = lst->last;
        if (last) {
            last->suc = p; /* add p after last */
            p->pre = last;
        }
        else {
            lst->first = p; /* p is the first element */
            p->pre = 0;
        }
    }
}

```

```

        }
        lst->last = p;           /* p is the new last element */
        p->suc = 0;
    }
}

```

However, we would never have gotten it right without drawing a few boxes and arrows on our doodle pad. Note that we “forgot” to consider the case where the argument **p** was null. Pass 0 instead of a pointer to a **Link** and this code will fail miserably. This is not inherently bad code, but it is *not* industrial strength. Its purpose is to illustrate common and useful techniques (and, in this case, also a common weakness/bug).

The **erase()** function can be written like this:

```

struct Link* erase(struct List* lst, struct Link* p)
/*
   remove p from lst;
   return a pointer to the link after p
*/
{
    assert(lst);
    if (p==0) return 0;           /* OK to erase(0) */

    if (p == lst->first) {
        if (p->suc) {
            lst->first = p->suc;      /* the successor becomes first */
            p->suc->pre = 0;
            return p->suc;
        }
        else {
            lst->first = lst->last = 0;  /* the list becomes empty */
            return 0;
        }
    }
    else if (p == lst->last) {
        if (p->pre) {
            lst->last = p->pre;      /* the predecessor becomes last */
            p->pre->suc = 0;
        }
        else {
            lst->first = lst->last = 0;  /* the list becomes empty */
            return 0;
        }
    }
}

```

```

    }
else {
    p->suc->pre = p->pre;
    p->pre->suc = p->suc;
    return p->suc;
}
}

```

We will leave the rest of the functions as an exercise, as we don't need them for our (all too simple) test. However, now we must face the central mystery of this design: Where is the data in the elements of the list? How do we implement a simple list of names represented by a C-style string? Consider:

```

struct Name {
    struct Link Link;      /* the Link required by List operations */
    char* p;              /* the name string */
};

```

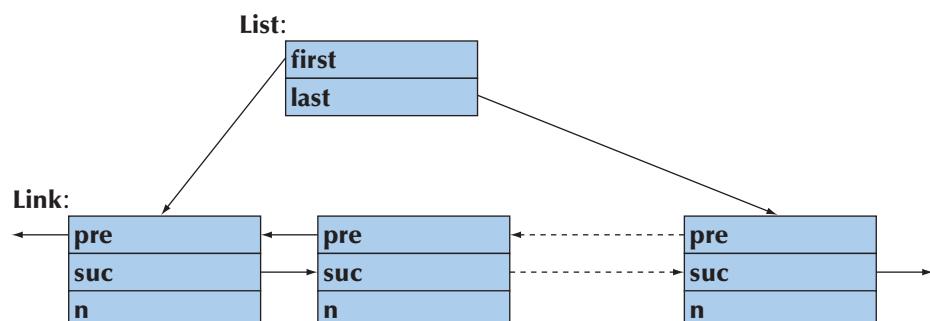
So far, so good, though how we get to use that **Link** member is a mystery; but since we know that a **List** likes its **Links** on the free store, we write a function creating **Names** on the free store:

```

struct Name* make_name(char* n)
{
    struct Name* p = (struct Name*)malloc(sizeof(struct Name));
    p->p = n;
    return p;
}

```

Or graphically:



Now let's use that:

```
int main()
{
    int count = 0;
    struct List names;           /* make a list */
    struct List* curr;
    init(&names);

    /* make a few Names and add them to the list: */
    push_back(&names,(struct Link*)make_name("Norah"));
    push_back(&names,(struct Link*)make_name("Annemarie"));
    push_back(&names,(struct Link*)make_name("Kris"));

    /* remove the second name (with index 1): */
    erase(&names,advance(names.first,1));

    curr = names.first;          /* write out all names */
    for (; curr!=0; curr=curr->suc) {
        count++;
        printf("element %d: %s\n", count, ((struct Name*)curr)->p);
    }
}
```

So we “cheated.” We used a cast to treat a **Name\*** as a **Link\***. In that way, the user knows about the “library-type” **Link**. However, the “library” doesn’t know about the “application-type” **Name**. Is that allowed? Yes, it is: in C (and C++), you can treat a pointer to a **struct** as a pointer to its first element and vice versa.

Obviously, this **List** example is also C++ exactly as written.

---

### TRY THIS



A common refrain among C++ programmers talking with C programmers is, “Everything you can do, I can do better!” So, rewrite the intrusive **List** example in C++, showing how to make it shorter and easier to use without making the code slower or the objects bigger.



## Drill

1. Write a “Hello, World!” program in C, compile it, and run it.
2. Define two variables holding “Hello” and “World!” respectively; concatenate them with a space in between; and output them as **Hello World!**.
3. Define a C function that takes a **char\*** parameter **p** and an **int** parameter **x** and print out their values in this format: **p is "foo" and x is 7**. Call it with a few argument pairs.

## Review

In the following, assume that by C we mean ISO standard C89.

1. Is C++ a subset of C?
2. Who invented C?
3. Name a highly regarded C textbook.
4. In what organization were C and C++ invented?
5. Why is C++ (almost) compatible with C?
6. Why is C++ only *almost* compatible with C?
7. List a dozen C++ features not present in C.
8. What organization “owns” C and C++?
9. List six C++ standard library components that cannot be used in C.
10. Which C standard library components can be used in C++?
11. How do you achieve function argument type checking in C?
12. What C++ features related to functions are missing in C? List at least three. Give examples.
13. How do you call a C function from C++?
14. How do you call a C++ function from C?
15. Which types are layout compatible between C and C++? (Just) give examples.
16. What is a structure tag?
17. List 20 C++ keywords that are not keywords in C.
18. Is **int x;** a definition in C++? In C?
19. What is a C-style cast and why is it dangerous?
20. What is **void\*** and how does it differ in C and C++?
21. How do enumerations differ in C and C++?
22. What do you do in C to avoid linkage problems from popular names?
23. What are the three most common C functions from free-store use?
24. What is the definition of a C-style string?
25. How do **==** and **strcmp()** differ for C-style strings?

26. How do you copy C-style strings?
27. How do you find the length of a C-style string?
28. How would you copy a large array of `ints`?
29. What's nice about `printf()`? What are its problems/limitations?
30. Why should you never use `gets()`? What can you use instead?
31. How do you open a file for reading in C?
32. What is the difference between `const` in C and `const` in C++?
33. Why don't we like macros?
34. What are common uses of macros?
35. What is an include guard?

## Terms

<code>#define</code>	Dennis Ritchie	non-intrusive
<code>#ifdef</code>	<code>FILE</code>	opaque type
<code>#ifndef</code>	<code>fopen()</code>	overloading
Bell Labs	format string	
Brian Kernighan	intrusive	
C/C++	K&R	
compatibility	lexicographical	
conditional compilation	linkage	three-way comparison
C-style cast	macro	
C-style string	<code>malloc()</code>	<code>void</code> <code>void*</code>

## Exercises

For these exercises it may be a good idea to compile all programs with both a C and a C++ compiler. If you use only a C++ compiler, you may accidentally use non-C features. If you use only a C compiler, type errors may remain undetected.

1. Implement versions of `strlen()`, `strcmp()`, and `strcpy()`.
2. Complete the intrusive `List` example in §27.9 and test it using every function.
3. “Pretty up” the intrusive `List` example in §27.9 as best you can to make it convenient to use. Do catch/handle as many errors as you can. It is fair game to change the details of the `struct` definitions, to use macros, whatever.
4. If you didn’t already, write a C++ version of the intrusive `List` example in §27.9 and test it using every function.
5. Compare the results of exercises 3 and 4.

6. Change the representation of **Link** and **List** from §27.9 without changing the user interface provided by the functions. Allocate **Links** in an array of links and have the members **first**, **last**, **pre**, and **suc** be **ints** (indices into the array).
7. What are the advantages and disadvantages of intrusive containers compared to C++ standard (non-intrusive) containers? Make lists of pros and cons.
8. What is the lexicographical order on your machine? Write out every character on your keyboard together with its integer value; then, write the characters out in the order determined by their integer value.
9. Using only C facilities, including the C standard library, read a sequence of words from **stdin** and write them to **stdout** in lexicographical order. Hint: The C sort function is called **qsort()**; look it up somewhere. Alternatively, insert the words into an ordered list as you read them. There is no C standard library list.
10. Make a list of C language features adopted from C++ or C with Classes (§27.1).
11. Make a list of C language features not adopted by C++.
12. Implement a (C-style **string**, **int**) lookup table with operations such as **find(struct table\*, const char\*)**, **insert(struct table\*, const char\*, int)**, and **remove(struct table\*, const char\*)**. The representation of the table could be an array of a **struct** pair or a pair of arrays (**const char\*[]** and **int\***); you choose. Also choose return types for your functions. Document your design decisions.
13. Write a program that does the equivalent of **string s; cin>>s;** in C; that is, define an input operation that reads an arbitrarily long sequence of whitespace-terminated characters into a zero-terminated array of **chars**.
14. Write a function that takes an array of **ints** as its input and finds the smallest and the largest elements. It should also compute the median and mean. Use a **struct** holding the results as the return value.
15. Simulate single inheritance in C. Let each “base class” contain a pointer to an array of pointers to functions (to simulate virtual functions as free-standing functions taking a pointer to a “base class” object as their first argument); see §27.2.3. Implement “derivation” by making the “base class” the type of the first member of the derived class. For each class, initialize the array of “virtual functions” appropriately. To test the ideas, implement a version of “the old **Shape** example” with the base and derived **draw()** just printing out the name of their class. Use only language features and library facilities available in standard C.
16. Use macros to obscure (simplify the notation for) the implementation in the previous exercise.

## Postscript

We did mention that compatibility issues are not all that exciting. However, there is a lot of C code “out there” (billions of lines of code), and if you have to read or write it, this chapter prepares you to do so. Personally, we prefer C++, and the information in this chapter gives part of the reason for that. And please don’t underestimate that “intrusive [List](#)” example – both “intrusive [Lists](#)” and opaque types are important and powerful techniques (in both C and C++).



# Part V

## Appendices





## Language Summary

“Be careful what you wish for;  
you might get it.”

—Traditional

This appendix summarizes key language elements of C++. The summary is very selective and specifically geared to novices who want to explore a bit beyond the sequence of topics in the book. The aim is conciseness, not completeness.

<b>A.1 General</b>	<b>A.9 Functions</b>
A.1.1 Terminology	A.9.1 Overload resolution
A.1.2 Program start and termination	A.9.2 Default arguments
A.1.3 Comments	A.9.3 Unspecified arguments
<b>A.2 Literals</b>	A.9.4 Linkage specifications
A.2.1 Integer literals	
A.2.2 Floating-point literals	
A.2.3 Boolean literals	
A.2.4 Character literals	
A.2.5 String literals	
A.2.6 The pointer literal	
<b>A.3 Identifiers</b>	
A.3.1 Keywords	
<b>A.4 Scope, storage class, and lifetime</b>	
A.4.1 Scope	
A.4.2 Storage class	
A.4.3 Lifetime	
<b>A.5 Expressions</b>	
A.5.1 User-defined operators	
A.5.2 Implicit type conversion	
A.5.3 Constant expressions	
A.5.4 <code>sizeof</code>	
A.5.5 Logical expressions	
A.5.6 <code>new</code> and <code>delete</code>	
A.5.7 Casts	
<b>A.6 Statements</b>	
<b>A.7 Declarations</b>	
A.7.1 Definitions	
<b>A.8 Built-in types</b>	
A.8.1 Pointers	
A.8.2 Arrays	
A.8.3 References	
	<b>A.10 User-defined types</b>
	A.10.1 Operator overloading
	<b>A.11 Enumerations</b>
	<b>A.12 Classes</b>
	A.12.1 Member access
	A.12.2 Class member definitions
	A.12.3 Construction, destruction, and copy
	A.12.4 Derived classes
	A.12.5 Bitfields
	A.12.6 Unions
	<b>A.13 Templates</b>
	A.13.1 Template arguments
	A.13.2 Template instantiation
	A.13.3 Template member types
	<b>A.14 Exceptions</b>
	<b>A.15 Namespaces</b>
	<b>A.16 Aliases</b>
	<b>A.17 Preprocessor directives</b>
	A.17.1 <code>#include</code>
	A.17.2 <code>#define</code>

## A.1 General

This appendix is a reference. It is not intended to be read from beginning to end like a chapter. It (more or less) systematically describes key elements of the C++ language. It is not a complete reference, though; it is just a summary. Its focus and emphasis were determined by student questions. Often, you will need to look at the chapters for a more complete explanation. This summary does not attempt to equal the precision and terminology of the standard. Instead, it attempts to be accessible. For more information, see Stroustrup, *The C++ Programming Language*.

The definition of C++ is the ISO C++ standard, but that document is neither intended for nor suitable for novices. Don't forget to use your online documentation. If you look at this appendix while working on the early chapters, expect much to be "mysterious," that is, explained in later chapters.

For standard library facilities, see Appendix B.

The standard for C++ is defined by a committee working under the auspices of the ISO (the international organization for standards) in collaboration with national standards bodies, such as INCITS (United States), BSI (United Kingdom), and AFNOR (France). The current definition is ISO/IEC 14882:2011 *Standard for Programming Language C++*.

### A.1.1 Terminology

The C++ standard defines what a C++ program is and what the various constructs mean:

- *Conforming*: A program that is C++ according to the standard is called *conforming* (or colloquially, *legal* or *valid*).
- *Implementation defined*: A program can (and usually does) depend on features (such as the size of an `int` and the numeric value of '`a`') that are only well defined on a given compiler, operating system, machine architecture, etc. The implementation-defined features are listed in the standard and must be documented in implementation documentation, and many are reflected in standard headers, such as `<limits>` (see §B.1.1). So, being conforming is not the same as being portable to all C++ implementations.
- *Unspecified*: The meaning of some constructs is *unspecified*, *undefined*, or *not conforming but not requiring a diagnostic*. Obviously, such features are best avoided. This book avoids them. The unspecified features to avoid include
  - Inconsistent definitions in separate source files (use header files consistently; see §8.3)
  - Reading *and* writing the same variable repeatedly in an expression (the main example is `a[i]=++i;`)
  - Many uses of explicit type conversion (casts), especially of `reinterpret_cast`

### A.1.2 Program start and termination

A C++ program must have a single global function called `main()`. The program starts by executing `main()`. The return type of `main()` is `int` (`void` is *not* a conforming alternative). The value returned by `main()` is the program's return value to "the system." Some systems ignore that value, but successful termination is indicated by returning zero and failure by returning a nonzero value or by an uncaught exception (but an uncaught exception is considered poor style).

The arguments to **main()** can be implementation defined, but every implementation must accept two versions (though only one per program):

```
int main();           // no arguments
int main(int argc, char* argv[]); // argv[] holds argc C-style strings
```

The definition of **main()** need not explicitly return a value. If it doesn't, "dropping through the bottom," it returns a zero. This is the minimal C++ program:

```
int main() { }
```

If you define a global (namespace) scope object with a constructor and a destructor, the constructor will logically be executed "before **main()**" and the destructor logically executed "after **main()**" (technically, executing those constructors is part of invoking **main()** and executing the destructors part of returning from **main()**). Whenever you can, avoid global objects, especially global objects requiring non-trivial construction and destruction.

### A.1.3 Comments

What can be said in code, should be. However, C++ offers two comment styles to allow the programmer to say things that are not well expressed as code:

```
// this is a line comment

/*
    this is a
    block comment
*/
```

Obviously, block comments are mostly used for multi-line comments, though some people prefer single-line comments even for multiple lines:

```
// this is a
// multi-line comment
// expressed using three line comments

/* and this is a single line of comment expressed using a block comment */
```

Comments are essential for documenting the intent of code; see also §7.6.4.

## A.2 Literals

Literals represent values of various types. For example, the literal **12** represents the integer value “twelve,” **"Morning"** represents the character string value *Morning*, and **true** represents the Boolean value *true*.

### A.2.1 Integer literals

*Integer literals* come in three varieties:

- Decimal: a series of decimal digits  
Decimal digits: **0, 1, 2, 3, 4, 5, 6, 7, 8**, and **9**
- Octal: a series of octal digits starting with **0**  
Octal digits: **0, 1, 2, 3, 4, 5, 6**, and **7**
- Hexadecimal: a series of hexadecimal digits starting with **0x** or **0X**  
Hexadecimal digits: **0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f, A, B, C, D, E**, and **F**
- Binary: a series of binary digits starting with **0b** or **0B** (C++14)  
Binary digits: **0, 1**

A suffix **u** or **U** makes an integer literal **unsigned** (§25.5.3), and a suffix **l** or **L** makes it **long**, for example, **10u** and **123456UL**.

C++14 also allows the use of the single quote as a digit separator in numeric literals. For example, **0b0000'0001'0010'0011** means **0b0000000100100011** and **1'000'000** means **1000000**.

#### A.2.1.1 Number systems

We usually write out numbers in decimal notation. **123** means **1** hundred plus **2** tens plus **3** ones, or **1\*100+2\*10+3\*1**, or (using **^** to mean “to the power of”) **1\*10^2+2\*10^1+3\*10^0**. Another word for *decimal* is *base-10*. There is nothing really special about 10 here. What we have is **1\*base^2+2\*base^1+3\*base^0** where **base==10**. There are lots of theories about why we use base-10. One theory has been “built into” some natural languages: we have ten fingers and each symbol, such as 0, 1, and 2, that directly stands for a value in a positional number system is called a digit. *Digit* is Latin for “finger.”

Occasionally, other bases are used. Typically, positive integer values in computer memory are represented in base-2 (it is relatively easy to reliably represent 0 and 1 as physical states in materials), and humans dealing with low-level hardware issues sometimes use base-8 and more often base-16 to refer to the content of memory.

Consider hexadecimal. We need to name the 16 values from 0 to 15. Usually, we use 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, where A has the decimal value 10, B the decimal value 11, and so on:

**A==10, B==11, C==12, D==13, E==14, F==15**

We can now write the decimal value **123** as **7B** using the hexadecimal notation. To see that, note that in the hexadecimal system **7B** means **7\*16+11**, which is (decimal) **123**. Conversely, hexadecimal **123** means **1\*16^2+2\*16+3**, which is **1\*256+2\*16+3**, which is (decimal) **291**. If you have never dealt with non-decimal integer representations, we strongly recommend you try converting a few numbers to and from decimal and hexadecimal. Note that a hexadecimal digit has a very simple correspondence to a binary value:

Hexadecimal and binary									
hex	0	1	2	3	4	5	6	7	
binary	0000	0001	0010	0011	0100	0101	0110	0111	
hex	8	9	A	B	C	D	E	F	
binary	1000	1001	1010	1011	1100	1101	1110	1111	

This goes a long way toward explaining the popularity of hexadecimal notation. In particular, the value of a byte is simply expressed as two hexadecimal digits.

In C++, (fortunately) numbers are decimal unless we specify otherwise. To say that a number is hexadecimal, we prefix **0X** (“X for hex”), so **123==0X7B** and **0X123==291**. We can equivalently use a lowercase **x**, so we also have **123==0x7B** and **0x123==291**. Similarly, we can use lowercase **a, b, c, d, e**, and **f** for the hexadecimal digits. For example, **123==0x7b**.

Octal is base-8. We need only eight octal digits: **0, 1, 2, 3, 4, 5, 6, 7**. In C++, base-8 numbers are represented starting with a **0**, so **0123** is not the decimal number **123**, but **1\*8^2+2\*8+3**, that is, **1\*64+2\*8+3**, or (decimal) **83**. Conversely, octal **83**, that is, **083**, is **8\*8+3**, which is (decimal) **67**. Using C++ notation, we get **0123==83** and **083==67**.

Binary is base-2. We need only two digits, **0** and **1**. We cannot directly represent base-2 numbers as literals in C++. Only base-8 (octal), base-10 (decimal), and base-16 (hexadecimal) are directly supported as literals and as input and output formats for integers. However, binary numbers are useful to know even if we cannot directly represent them in C++ text. For example, (decimal) **123** is

$$1*2^6+1*2^5+1*2^4+1*2^3+0*2^2+1*2^1$$

which is  $1*64+1*32+1*16+1*8+0*4+1*2+1$ , which is (binary) **1111011**.

### A.2.2 Floating-point-literals

A *floating-point-literal* contains a decimal point (.), an exponent (e.g., **e3**), or a floating-point suffix (**d** or **f**). For example:

<b>123</b>	// int (no decimal point, suffix, or exponent)
<b>123.</b>	// double: 123.0
<b>123.0</b>	// double
<b>123</b>	// double: 0.123
<b>0.123</b>	// double
<b>1.23e3</b>	// double: 1230.0
<b>1.23e-3</b>	// double: 0.00123
<b>1.23e+3</b>	// double: 1230.0

Floating-point-literals have type **double** unless a suffix indicates otherwise. For example:

<b>1.23</b>	// double
<b>1.23f</b>	// float
<b>1.23L</b>	// long double

### A.2.3 Boolean literals

The literals of type **bool** are **true** and **false**. The integer value of **true** is **1** and the integer value of **false** is **0**.

### A.2.4 Character literals

A *character literal* is a character enclosed in single quotes, for example, '**a**' and '**@**'. In addition, there are some "special characters":

Name	ASCII name	C++ name
newline	<b>NL</b>	<b>\n</b>
horizontal tab	<b>HT</b>	<b>\t</b>
vertical tab	<b>VT</b>	<b>\v</b>
backspace	<b>BS</b>	<b>\b</b>
carriage return	<b>CR</b>	<b>\r</b>
form feed	<b>FF</b>	<b>\f</b>
alert	<b>BEL</b>	<b>\a</b>

Name	ASCII name	C++ name
backslash	\	\\\
question mark	?	\?
single quote	'	\'
double quote	"	\\"
octal number	ooo	\ooo
hexadecimal number	hhh	\xhhh

A special character is represented as its “C++ name” enclosed in single quotes, for example, '\n' (newline) and '\t' (tab).

The character set includes the following visible characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!@#$%^&*()_+|~{}[]:;'<>?,./
```

In portable code, you cannot rely on more visible characters. The value of a character, such as 'a' for a, is implementation dependent (but easily discovered, for example, `cout << int('a')`).

### A.2.5 String literals

A *string literal* is a series of characters enclosed in double quotes, for example, "Knuth" and "King Canute". A newline cannot be part of a string; instead use the special character '\n' to represent newline in a string:

```
"King
Canute "           // error: newline in string literal
"King\nCanute"     // OK: correct way to get a newline into a string literal
```

Two string literals separated only by whitespace are taken as a single string literal. For example:

```
"King" "Canute"    // equivalent to "KingCanute" (no space)
```

Note that special characters, such as '\n', can appear in string literals.

### A.2.6 The pointer literal

There is only one *pointer literal*: the null pointer, `nullptr`. For compatibility, any constant expression that evaluates to `0` can also be used as the null pointer. For example:

```
t* p1 = 0;           // OK: null pointer
int* p2 = 2-2;       // OK: null pointer
int* p3 = 1;         // error: 1 is an int, not a pointer
int z = 0;
int* p4 = z;         // error: z is not a constant
```

The value `0` is implicitly converted to the null pointer.

In C++ (but not in C, so beware of C headers), `NULL` is defined to mean `0` so that you can write

```
int* p4 = NULL;     // (given the right definition of NULL) the null pointer
```

## A.3 Identifiers

An *identifier* is a sequence of characters starting with a letter or an underscore followed by zero or more (uppercase or lowercase) letters, digits, or underscores:

```
int foo_bar;          // OK
int FooBar;           // OK
int foo bar;          // error: space can't be used in an identifier
int foo$bar;           // error: $ can't be used in an identifier
```

Identifiers starting with an underscore or containing a double underscore are reserved for use by the implementation; don't use them. For example:

```
int _foo;              // don't
int foo_bar;            // OK
int foo__bar;           // don't
int foo_;                // OK
```

### A.3.1 Keywords

*Keywords* are identifiers used by the language itself to express language constructs.

Keywords (reserved identifiers)					
alignas	class	explicit	noexcept	signed	typename
alignof	compl	export	not	sizeof	union
and	concept	extern	not_eq	static	unsigned
and_eq	const	false	nullptr	static_assert	using
asm	const_cast	float	operator	static_cast	virtual
auto	constexpr	for	or	struct	void
bitand	continue	friend	or_eq	switch	volatile
bitor	decltype	goto	private	template	wchar_t
bool	default	if	protected	this	while
break	delete	inline	public	thread_local	xor
case	do	int	register	throw	xor_eq
catch	double	long	reinterpret_cast	true	
char	dynamic_cast	mutable	requires	try	
char16_t	else	namespace	return	typedef	
char32_t	enum	new	short	typeid	

## A.4 Scope, storage class, and lifetime

Every name in C++ (with the lamentable exception of preprocessor names; see §A.17) exists in a scope; that is, the name belongs to a region of text in which it can be used. Data (objects) are stored in memory somewhere; the kind of memory used to store an object is called its *storage class*. The lifetime of an object is from the time it is first initialized until it is finally destroyed.

### A.4.1 Scope

There are five kinds of *scopes* (§8.4):

- *Global scope:* A name is in global scope unless it is declared inside some language construct (e.g., a class or a function).
- *Namespace scope:* A name is in a namespace scope if it is defined within a namespace and not inside some language construct (e.g., a class or a function). Technically, the global scope is a namespace scope with “the empty name.”

- *Local scope*: A name is in a local scope if it is declared inside a function (this includes function parameters).
- *Class scope*: A name is in a class scope if it is the name of a member of a class.
- *Statement scope*: A name is in a statement scope if it is declared in the (...) part of a **for**-, **while**-, **switch**-, or **if**-statement.

The scope of a variable extends (only) to the end of the statement in which it is defined. For example:

```
for (int i = 0; i < v.size(); ++i) {
    // i can be used here
}
if (i < 27)           // the i from the for-statement is not in scope here
```

Class and namespace scopes have names, so that we can refer to a member from “elsewhere.” For example:

```
void f();           // in global scope

namespace N {
    void f()        // in namespace scope N
    {
        int v;      // in local scope
        ::f();       // call the global f()
    }
}

void f()
{
    N::f();        // call N's f()
}
```

What would happen if you called **N::f()** or **::f()**? See also §A.15.

### A.4.2 Storage class

There are three *storage classes* (§17.4):

- *Automatic storage*: Variables defined in functions (including function parameters) are placed in automatic storage (i.e., “on the stack”) unless explicitly declared to be **static**. Automatic storage is allocated when a function is called and deallocated when a call returns; thus, if a function is (directly)

or indirectly) called by itself, multiple copies of automatic data can exist: one for each call (§8.5.8).

- *Static storage*: Variables declared in global and namespace scope are stored in static storage, as are variables explicitly declared **static** in functions and classes. The linker allocates static storage “before the program starts running.”
- *Free store (heap)*: Objects created by **new** are allocated in the free store.

For example:

```
vector<int> vg(10);      // constructed once at program start ("before main()")

vector<int>* f(int x)
{
    static vector<int> vs(x); // constructed in first call of f() only
    vector<int> vf(x+x);    // constructed in each call of f()

    for (int i=1; i<10; ++i) {
        vector<int> v1(i);   // constructed in each iteration
        // ...
    }                         // v1 destroyed here (in each iteration)

    return new vector<int>(vf); // constructed on free store as a copy of vf
}                           // vf destroyed here

void ff()
{
    vector<int>* p = f(10); // get vector from f()
    // ...
    delete p;             // delete the vector from f
}
```

The statically allocated variables **vg** and **vs** are destroyed at program termination (“after **main()**”), provided they have been constructed.

Class members are not allocated as such. When you allocate an object somewhere, the non-static members are placed there also (with the same storage class as the class object to which they belong).

Code is stored separately from data. For example, a member function is *not* stored in each object of its class; one copy is stored with the rest of the code for the program.

See also §14.3 and §17.4.

### A.4.3 Lifetime

Before an object can be (legally) used, it must be initialized. This initialization can be explicit using an initializer or implicit using a constructor or a rule for default initialization of built-in types. The lifetime of an object ends at a point determined by its scope and storage class (e.g., see §17.4 and §B.4.2):

- *Local (automatic) objects* are constructed if/when the thread of execution gets to them and are destroyed at end of scope.
- *Temporary objects* are created by a specific sub-expression and destroyed at the end of their full expression. A full expression is an expression that is not a sub-expression of some other expression.
- *Namespace objects and static class members* are constructed at the start of the program (“before **main()**”) and destroyed at the end of the program (“after **main()**”).
- *Local static objects* are constructed if/when the thread of execution gets to them and (if constructed) are destroyed at the end of the program.
- *Free-store objects* are constructed by **new** and optionally destroyed using **delete**.

A temporary variable bound to a local or namespace reference “lives” as long as the reference. For example:

```
const char* string_tbl[] = { "Mozart", "Grieg", "Haydn", "Chopin" };
const char* f(int i) { return string_tbl[i]; }
void g(string s){}

void h()
{
    const string& r = f(0);      // bind temporary string to r
    g(f(1));                  // make a temporary string and pass it
    string s = f(2);            // initialize s from temporary string
    cout << "f(3): " << f(3)    // make a temporary string and pass it
        << " s: " << s
        << " r: " << r << '\n';
}
```

The result is

**f(3): Chopin s: Haydn r: Mozart**

The **string** temporaries generated for the calls **f(1)**, **f(2)**, and **f(3)** are destroyed at the end of the expression in which they were created. However, the temporary generated for **f(0)** is bound to **r** and “lives” until the end of **h()**.

## A.5 Expressions

This section summarizes C++’s operators. We use abbreviations that we find mnemonic, such as **m** for a member name, **T** for a type name, **p** for an expression yielding a pointer, **x** for an expression, **v** for an lvalue expression, and **lst** for an argument list. The result type of the arithmetic operations is determined by “the usual arithmetic conversions” (§A.5.2.2). The descriptions in this section are of the built-in operators, not of any operator you might define on your own, though when you define your own operators, you are encouraged to follow the semantic rules described for built-in operations (§9.6).

### Scope resolution

**N :: m**    **m** is in the namespace **N**; **N** is the name of a namespace or a class.

**:: m**    **m** is in the global namespace.

Note that members can themselves nest, so that you can get **N::C::m**; see also §8.7.

### Postfix expressions

<b>x . m</b>	member access; <b>x</b> must be a class object
<b>p -&gt; m</b>	member access; <b>p</b> must point to a class object; equivalent to <b>(*p).m</b>
<b>p[x]</b>	subscripting; equivalent to <b>*(p+x)</b>
<b>f(lst)</b>	function call: call <b>f</b> with the argument list <b>lst</b>
<b>T(lst)</b>	construction: construct a <b>T</b> with the argument list <b>lst</b>
<b>v++</b>	(post-)increment; the value of <b>v++</b> is the value of <b>v</b> before incrementing
<b>v--</b>	(post-)decrement; the value of <b>v--</b> is the value of <b>v</b> before decrementing
<b>typeid(x)</b>	run-time type identification for <b>x</b>
<b>typeid(T)</b>	run-time type identification for <b>T</b>
<b>dynamic_cast&lt;T&gt;(x)</b>	run-time checked conversion of <b>x</b> to <b>T</b>
<b>static_cast&lt;T&gt;(x)</b>	compile-time checked conversion of <b>x</b> to <b>T</b>
<b>const_cast&lt;T&gt;(x)</b>	unchecked conversion to add or remove <b>const</b> from <b>x</b> ’s type to get <b>T</b>
<b>reinterpret_cast&lt;T&gt;(x)</b>	unchecked conversion of <b>x</b> to <b>T</b> by reinterpreting the bit pattern of <b>x</b>

The `typeid` operator and its uses are not covered in this book; see an expert-level reference. Note that casts do not modify their argument. Instead, they produce a result of their type, which somehow corresponds to the argument value; see §A.5.7.

Unary expressions	
<code>sizeof(T)</code>	the size of a <code>T</code> in bytes
<code>sizeof(x)</code>	the size of an object of <code>x</code> 's type in bytes
<code>++v</code>	(pre-)increment; equivalent to <code>v+=1</code>
<code>--v</code>	(pre-)decrement; equivalent to <code>v-=1</code>
<code>~x</code>	complement of <code>x</code> ; <code>~</code> is a bitwise operation
<code>!x</code>	not <code>x</code> ; returns <code>true</code> or <code>false</code>
<code>&amp;v</code>	address of <code>v</code>
<code>*p</code>	contents of object pointed to by <code>p</code>
<code>new T</code>	make a <code>T</code> on the free store
<code>new T(lst)</code>	make a <code>T</code> on the free store and initialize it with <code>lst</code>
<code>new(lst) T</code>	construct a <code>T</code> at the location determined by <code>lst</code>
<code>new(lst) T(lst2)</code>	construct a <code>T</code> at the location determined by <code>lst</code> and initialize it with <code>lst2</code>
<code>delete p</code>	free the object pointed to by <code>p</code>
<code>delete[] p</code>	free the array of objects pointed to by <code>p</code>
<code>(T)x</code>	C-style cast; convert <code>x</code> to <code>T</code>

Note that the object(s) pointed to by `p` in `delete p` and `delete[] p` must be allocated using `new`; see §A.5.6. Note that `(T)x` is far less specific – and therefore more error-prone – than the more specific cast operators; see §A.5.7.

Member selection	
<code>x.*ptm</code>	the member of <code>x</code> identified by the pointer-to-member <code>ptm</code>
<code>p-&gt;*ptm</code>	the member of <code>*p</code> identified by the pointer-to-member <code>ptm</code>

Not covered in this book; see an expert-level reference.

### Multiplicative operators

<b>x*y</b>	Multiply <b>x</b> by <b>y</b> .
<b>x/y</b>	Divide <b>x</b> by <b>y</b> .
<b>x%y</b>	Modulo (remainder) of <b>x</b> by <b>y</b> (not for floating-point types).

The effect of **x/y** and **x%y** is undefined if **y==0**. The effect of **x%y** is implementation defined if **x** or **y** is negative.

### Additive operators

<b>x+y</b>	Add <b>x</b> and <b>y</b> .
<b>x-y</b>	Subtract <b>y</b> from <b>x</b> .

### Shift operators

<b>x&lt;&lt;y</b>	Shift <b>x</b> left by <b>y</b> bit positions.
<b>x&gt;&gt;y</b>	Shift <b>x</b> right by <b>y</b> bit positions.

For the (built-in) use of **>>** and **<<** for shifting bits, see §25.5.4. When their left-most operators are **iostreams**, these operators are used for I/O; see Chapters 10 and 11.

### Relational operators

<b>x&lt;y</b>	<b>x</b> less than <b>y</b> ; returns a <b>bool</b>
<b>x&lt;=y</b>	<b>x</b> less than or equal to <b>y</b>
<b>x&gt;y</b>	<b>x</b> greater than <b>y</b>
<b>x&gt;=y</b>	<b>x</b> greater than or equal to <b>y</b>

The result of a relational operator is a **bool**.

### Equality operators

<b>x==y</b>	<b>x</b> equals <b>y</b> ; returns a <b>bool</b>
<b>x!=y</b>	<b>x</b> not equal to <b>y</b>

Note that **x!=y** is **!(x==y)**. The result of an equality operator is a **bool**.

**Bitwise and**

<b>x&amp;y</b>	bitwise and of <b>x</b> and <b>y</b>
----------------	--------------------------------------

Note that **&** (like **^**, **|**, **~**, **>>**, and **<<**) delivers a set of bits. For example, if **a** and **b** are **unsigned chars**, **a&b** is an **unsigned char** with each bit being the result of applying **&** to the corresponding bits in **a** and **b**; see §A.5.5.

**Bitwise xor**

<b>x^y</b>	bitwise exclusive or of <b>x</b> and <b>y</b>
------------	---

**Bitwise or**

<b>x y</b>	bitwise or of <b>x</b> and <b>y</b>
------------	-------------------------------------

**Logical and**

<b>x&amp;&amp;y</b>	logical and; returns <b>true</b> or <b>false</b> ; evaluate <b>y</b> only if <b>x</b> is true
---------------------	---

**Logical or**

<b>x  y</b>	logical or; returns <b>true</b> or <b>false</b> ; evaluate <b>y</b> only if <b>x</b> is false
-------------	---

See §A.5.5.

**Conditional expression**

<b>x?y:z</b>	If <b>x</b> the result is <b>y</b> ; otherwise the result is <b>z</b> .
--------------	---

For example:

```
template<class T> T& max(T& a, T& b) { return (a>b)?a:b; }
```

The “question mark colon operator” is explained in §8.4.

**Assignments**

<b>v=x</b>	assign <b>x</b> to <b>v</b> ; result is the resulting <b>v</b>
------------	--

<b>v*=x</b>	roughly <b>v=v*(x)</b>
-------------	------------------------

### Assignments (*continued*)

<code>v/=x</code>	roughly <code>v=v/(x)</code>
<code>v%=/x</code>	roughly <code>v=v%/(x)</code>
<code>v+=x</code>	roughly <code>v=v+(x)</code>
<code>v-=x</code>	roughly <code>v=v-(x)</code>
<code>v&gt;&gt;=x</code>	roughly <code>v=v&gt;&gt;(x)</code>
<code>v&lt;&lt;=x</code>	roughly <code>v=v&lt;&lt;(x)</code>
<code>v&amp;=x</code>	roughly <code>v=v&amp;(x)</code>
<code>v^=x</code>	roughly <code>v=v^*(x)</code>
<code>v =x</code>	roughly <code>v=v (x)</code>

By “roughly `v=v*(x)`” we mean that `v*=x` has that value except that `v` is evaluated only once. For example, `v[++i]*=7+3` means `(++i, v[i]=v[i]*(7+3))` rather than `(v[++i]=v[++i]*(7+3))` (which would be undefined; see §8.6.1).

### Throw expression

<code>throw x</code>	Throw the value of <code>x</code> .
----------------------	-------------------------------------

The type of a `throw` expression is `void`.

### Comma expression

<code>x,y</code>	Execute <code>x</code> then <code>y</code> ; the result is <code>y</code> .
------------------	---

Each box holds operators with the same precedence. Operators in higher boxes have higher precedence than operators in lower boxes. For example, `a+b*c` means `a+(b*c)` rather than `(a+b)*c` because `*` has higher precedence than `+`. Similarly, `*p++` means `*(p++)`, not `(*p)++`. Unary operators and assignment operators are right-associative; all others are left-associative. For example, `a=b=c` means `a=(b=c)` and `a+b+c` means `(a+b)+c`.

An lvalue is an expression that identifies an object that could in principle be modified (but obviously an lvalue that has a `const` type is protected against modification by the type system) and have its address taken. The complement to lvalue is rvalue, that is, an expression that identifies something that may not be modified or have its address taken, such as a value returned from a function (`&f(x)` is an error because `f(x)` is an rvalue).

### A.5.1 User-defined operators

The rules defined here are for built-in types. If a user-defined operator is used, an expression is simply transformed into a call of the appropriate user-defined operator function, and the rules for function call determine what happens. For example:

```
class Mine { /* ... */;
bool operator==(Mine, Mine);

void f(Mine a, Mine b)
{
    if (a==b) {      // a==b means operator==(a,b)
        // ...
    }
}
```

A user-defined type is a class (§A.12, Chapter 9) or an enumeration (§A.11, §9.5).

### A.5.2 Implicit type conversion

Integral and floating-point types (§A.8) can be mixed freely in assignments and expressions. Wherever possible, values are converted so as not to lose information. Unfortunately, value-destroying conversions are also performed implicitly.

#### A.5.2.1 Promotions

The implicit conversions that preserve values are commonly referred to as *promotions*. Before an arithmetic operation is performed, *integral promotion* is used to create **ints** out of shorter integer types. This reflects the original purpose of these promotions: to bring operands to the “natural” size for arithmetic operations. In addition, **float** to **double** is considered a promotion.

Promotions are used as part of the usual arithmetic conversions (see §A.5.2.2).

#### A.5.2.2 Conversions

The fundamental types can be converted into each other in a bewildering number of ways. When writing code, you should always aim to avoid undefined behavior and conversions that quietly throw away information (see §3.9 and §25.5.3). A compiler can warn about many questionable conversions.

- *Integral conversions*: An integer can be converted to another integer type. An enumeration value can be converted to an integer type. If the destination type is **unsigned**, the resulting value is simply as many bits from the source as will fit in the destination (high-order bits are thrown away if necessary). If the destination type is signed, the value is unchanged if it

can be represented in the destination type; otherwise, the value is implementation defined. Note that **bool** and **char** are integer types.

- *Floating-point conversions:* A floating-point value can be converted to another floating-point type. If the source value can be exactly represented in the destination type, the result is the original numeric value. If the source value is between two adjacent destination values, the result is one of those values. Otherwise, the behavior is undefined. Note that **float** to **double** is considered a promotion.
- *Pointer and reference conversions:* Any pointer to an object type can be implicitly converted to a **void\*** (§17.8, §27.3.5). A pointer (reference) to a derived class can be implicitly converted to a pointer (reference) to an accessible and unambiguous base (§14.3). A constant expression (§A.5, §4.3.1) that evaluates to 0 can be implicitly converted to any pointer type. A **T\*** can be implicitly converted to a **const T\***. Similarly, a **T&** can be implicitly converted to a **const T&**.
- *Boolean conversions:* Pointers, integrals, and floating-point values can be implicitly converted to **bool**. A nonzero value converts to **true**; a zero value converts to **false**.
- *Floating-to-integer conversions:* When a floating-point value is converted to an integer value, the fractional part is discarded. In other words, conversion from a floating-point type to an integer type truncates. The behavior is undefined if the truncated value cannot be represented in the destination type. Conversions from integer to floating types are as mathematically correct as the hardware allows. Loss of precision occurs if an integral value cannot be represented exactly as a value of the floating type.
- *Usual arithmetic conversions:* These conversions are performed on the operands of a binary operator to bring them to a common type, which is then used as the type of the result:
  1. If either operand is of type **long double**, the other is converted to **long double**. Otherwise, if either operand is **double**, the other is converted to **double**. Otherwise, if either operand is **float**, the other is converted to **float**. Otherwise, integral promotions are performed on both operands.
  2. Then, if either operand is **unsigned long**, the other is converted to **unsigned long**. Otherwise, if one operand is a **long int** and the other is an **unsigned int**, then if a **long int** can represent all the values of an **unsigned int**, the **unsigned int** is converted to a **long int**; otherwise, both operands are converted to **unsigned long int**. Otherwise, if either operand is **long**, the other is converted to **long**. Otherwise, if either operand is **unsigned**, the other is converted to **unsigned**. Otherwise, both operands are **int**.

Obviously, it is best not to rely too much on complicated mixtures of types, so as to minimize the need for implicit conversions.

### A.5.2.3 User-defined conversions

In addition to the standard promotions and conversions, a programmer can define conversions for user-defined types. A constructor that takes a single argument defines a conversion from its argument type to its type. If the constructor is **explicit** (see §18.4.1), the conversion happens only when the programmer explicitly requires the conversion. Otherwise, the conversion can be implicit.

### A.5.3 Constant expressions

A *constant expression* is an expression that can be evaluated at compile time. For example:

```
const int a = 2.7*3;
const int b = a+3;
```

```
constexpr int a = 2.7*3;
constexpr int b = a+3;
```

A **const** can be initialized with an expression involving variables. A **constexpr** must be initialized by a constant expression. Constant expressions are required in a few places, such as array bounds, case labels, enumerator initializers, and **int** template arguments. For example:

```
int var = 7;
switch (x) {
    case 77:      // OK
    case a+2:     // OK
    case var:     // error (var is not a constant expression)
        ...
};
```

A function declared **constexpr** can be used in a constant expression.

### A.5.4 **sizeof**

In **sizeof(x)**, **x** can be a type or an expression. If **x** is an expression, the value of **sizeof(x)** is the size of the resulting object. If **x** is a type, **sizeof(x)** is the size of an object of type **x**. Sizes are measured in bytes. By definition, **sizeof(char)==1**.

### A.5.5 Logical expressions

C++ provides logical operators for integer types:

Bitwise logical operations	
<code>x&amp;y</code>	bitwise and of <code>x</code> and <code>y</code>
<code>x y</code>	bitwise or of <code>x</code> and <code>y</code>
<code>x^y</code>	bitwise exclusive or of <code>x</code> and <code>y</code>

Logical operations	
<code>x&amp;&amp;y</code>	logical and; returns <code>true</code> or <code>false</code> ; evaluate <code>y</code> only if <code>x</code> is <code>true</code>
<code>x  y</code>	logical or; returns <code>true</code> or <code>false</code> ; evaluate <code>y</code> only if <code>x</code> is <code>false</code>

The bitwise operators do their operation on each bit of their operands, whereas the logical operators (`&&` and `||`) treat a `0` as the value `false` and anything else as the value `true`. The definitions of the operations are:

<code>&amp;</code>	<code>0</code>	<code>1</code>	<code> </code>	<code>0</code>	<code>1</code>	<code>^</code>	<code>0</code>	<code>1</code>
<code>0</code>	0	0	<code>0</code>	0	1	<code>0</code>	0	1
<code>1</code>	0	1	<code>1</code>	1	1	<code>1</code>	1	0

### A.5.6 new and delete

Memory on the free store (dynamic store, heap) is allocated using `new` and deallocated (“freed”) using `delete` (for individual objects) or `delete[]` (for an array). If memory is exhausted, `new` throws a `bad_alloc` exception. A successful `new` operation allocates at least 1 byte and returns a pointer to the allocated object. The type of object allocated is specified after `new`. For example:

```
int* p1 = new int;           // allocate an (uninitialized) int
int* p2 = new int(7);        // allocate an int initialized to 7
int* p3 = new int[100];      // allocate 100 (uninitialized) ints
// ...
delete p1;                  // deallocate individual object
delete p2;
delete[] p3;                // deallocate array
```

If you allocate objects of a built-in type using `new`, they will not be initialized unless you specify an initializer. If you allocate objects of a class with a constructor using `new`, a constructor is called; the default constructor is called unless you specify an initializer (§17.4.4).

A `delete` invokes the destructor, if any, for its operand. Note that a destructor may be virtual (§A.12.3.1).

### A.5.7 Casts

There are four type-conversion operators:

Type-conversion operators	
<code>x=dynamic_cast&lt;D*&gt;(p)</code>	Try to convert <code>p</code> into a <code>D*</code> (may return <code>0</code> ).
<code>x=dynamic_cast&lt;D&amp;&gt;(*p)</code>	Try to convert <code>*p</code> into a <code>D&amp;</code> (may throw <code>bad_cast</code> ).
<code>x=static_cast&lt;T&gt;(v)</code>	Convert <code>v</code> into a <code>T</code> if a <code>T</code> can be converted into <code>v</code> 's type.
<code>x=reinterpret_cast&lt;T&gt;(v)</code>	Convert <code>v</code> into a <code>T</code> represented by the same bit pattern.
<code>x=const_cast&lt;T&gt;(v)</code>	Convert <code>v</code> into a <code>T</code> by adding or subtracting <code>const</code> .
<code>x=(T)v</code>	C-style cast: do any old cast.
<code>x=T(v)</code>	Functional cast: do any old cast.
<code>X=T{v}</code>	Construct a <code>T</code> from <code>v</code> (no narrowing will be done).

The dynamic cast is typically used for class hierarchy navigation where `p` is a pointer to a base class and `D` is derived from that base. It returns `0` if `v` is not a `D*`. If you want `dynamic_cast` to throw an exception (`bad_cast`) instead of returning `0`, cast to a reference instead of to a pointer. The dynamic cast is the only cast that relies on run-time checking.

Static cast is used for “reasonably well-behaved conversions,” that is, where `v` could have been the result of an implicit conversion from a `T`; see §17.8.

Reinterpret cast is used for reinterpreting a bit pattern. It is not guaranteed to be portable. In fact, it is best to assume that every use of `reinterpret_cast` is non-portable. A typical example is an `int`-to-pointer conversion to get a machine address into a program; see §17.8 and §25.4.1.

The C-style and functional casts can perform any conversion that can be achieved by a `static_cast` or a `reinterpret_cast`, combined with a `const_cast`.

Casts are best avoided. In most cases, consider their use a sign of poor programming. Exceptions to this rule are presented in §17.8 and §25.4.1. The C-style cast and function-style casts have the nasty property that you don't have to understand exactly what the cast is doing (§27.3.4). Prefer the named casts when you cannot avoid an explicit type conversion.

## A.6 Statements

Here is a grammar for C++'s statements (*opt* means “optional”):

*statement*:

- declaration*
- { *statement-list<sub>opt</sub>* }
- try** { *statement-list<sub>opt</sub>* } *handler-list*
- expression<sub>opt</sub>* ;
- selection-statement*
- iteration-statement*
- labeled-statement*
- control-statement*

*selection-statement*:

- if** ( *condition* ) *statement*
- if** ( *condition* ) *statement* **else** *statement*
- switch** ( *condition* ) *statement*

*iteration-statement*:

- while** ( *condition* ) *statement*
- do** *statement* **while** ( *expression* ) ;
- for** ( *for-init-statement* *condition<sub>opt</sub>* ; *expression<sub>opt</sub>* ) *statement*
- for** ( *declaration* : *expression* ) *statement*

*labeled-statement*:

- case** *constant-expression* : *statement*
- default** : *statement*
- identifier** : *statement*

*control-statement*:

- break** ;
- continue** ;
- return** *expression<sub>opt</sub>* ;
- goto** *identifier* ;

*statement-list*:

- statement* *statement-list<sub>opt</sub>*

*condition*:

- expression*
- type-specifier declarator* = *expression*

*for-init-statement:*

*expression<sub>opt</sub> ;*  
*type-specifier declarator = expression ;*

*handler-list:*

**catch** ( *exception-declaration* ) { *statement-list<sub>opt</sub>* }  
*handler-list handler-list<sub>opt</sub>*

Note that a declaration is a statement and that there is no assignment statement or procedure call statement; assignments and function calls are expressions. More information:

- Iteration (**for** and **while**); see §4.4.2.
- Selection (**if**, **switch**, **case**, and **break**); see §4.4.1. A **break** “breaks out of” the nearest enclosing **switch**-statement, **while**-statement, **do**-statement, or **for**-statement; that is, the next statement executed will be the statement following that enclosing statement.
- Expressions; see §A.5, §4.3.
- Declarations; see §A.6, §8.2.
- Exceptions (**try** and **catch**); see §5.6, §19.4.

Here is an example concocted simply to demonstrate a variety of statements (what does it do?):

```
int* f(int p[], int n)
{
    if (p==0) throw Bad_p(n);
    vector<int> v;
    int x;
    while (cin>>x) {
        if (x==terminator) break;    // exit while loop
        v.push_back(x);
    }
    for (int i = 0; i<v.size() && i<n; ++i) {
        if (v[i]==*p)
            return p;
        else
            ++p;
    }
    return 0;
}
```

## A.7 Declarations

A *declaration* consists of three parts:

- The name of the entity being declared
- The type of the entity being declared
- The initial value of the entity being declared (optional in most cases)

We can declare

- Objects of built-in types and user-defined types (§A.8)
- User-defined types (classes and enumerations) (§A.10–11, Chapter 9)
- Templates (class templates and function templates) (§A.13)
- Aliases (§A.16)
- Namespaces (§A.15, §8.7)
- Functions (including member functions and operators) (§A.9, Chapter 8)
- Enumerators (values for enumerations) (§A.11, §9.5)
- Macros (§A.17.2, §27.8)

The initializer can be a `{ }`-delimited list of expressions with zero or more elements (§3.9.2, §9.4.2, §18.2). For example:

```
vector<int> v {a,b,c,d};  
int x {y*z};
```

If the type of the object in a definition is **auto**, the object must be initialized and the type is the type of the initializer (§13.3, §21.2). For example:

```
auto x = 7;           // x is an int  
const auto pi = 3.14; // pi is a double  
for (const auto& x : v) // x is a reference to an element of v
```

### A.7.1 Definitions

A declaration that initializes, sets aside memory, or in other ways provides all the information necessary for using a name in a program is called a *definition*. Each type, object, and function in a program must have exactly one definition. Examples:

```
double f();           // a declaration  
double f() { /* ... */}; // (also) a definition  
extern const int x;   // a declaration
```

```
int y;           // (also) a definition
int z = 10;      // a definition with an explicit initializer
```

A **const** must be initialized. This is achieved by requiring an initializer for a **const** unless it has an explicit **extern** in its declaration (so that the initializer must be on its definition elsewhere) or it is of a type with a default constructor (§A.12.3). Class members that are **consts** must be initialized in every constructor using a member initializer (§A.12.3).

## A.8 Built-in types

C++ has a host of fundamental types and types constructed from fundamental types using modifiers:

Built-in types	
<b>bool x</b>	<b>x</b> is a Boolean (values <b>true</b> and <b>false</b> ).
<b>char x</b>	<b>x</b> is a character (usually 8 bits).
<b>short x</b>	<b>x</b> is a short <b>int</b> (usually 16 bits).
<b>int x</b>	<b>x</b> is of the default integer type.
<b>float x</b>	<b>x</b> is a floating-point number (a “short double”).
<b>double x</b>	<b>x</b> is a (“double-precision”) floating-point number.
<b>void* p</b>	<b>p</b> is a pointer to raw memory (memory of unknown type).
<b>T* p</b>	<b>p</b> is a pointer to <b>T</b> .
<b>T *const p</b>	<b>p</b> is a constant (immutable) pointer to <b>T</b> .
<b>T a[n]</b>	<b>a</b> is an array of <b>n T</b> s.
<b>T&amp; r</b>	<b>r</b> is a reference to <b>T</b> .
<b>T f(arguments)</b>	<b>f</b> is a function taking <b>arguments</b> and returning a <b>T</b> .
<b>const T x</b>	<b>x</b> is a constant (immutable) version of <b>T</b> .
<b>long T x</b>	<b>x</b> is a <b>long T</b> .
<b>unsigned T x</b>	<b>x</b> is an <b>unsigned T</b> .
<b>signed T x</b>	<b>x</b> is a <b>signed T</b> .

Here, **T** indicates “some type,” so you can have a **long unsigned int**, a **long double**, an **unsigned char**, and a **const char \*** (pointer to constant **char**). However, this system is not perfectly general; for example, there is no **short double** (that would have been a **float**), no **signed bool** (that would have been meaningless), no **short**

**long int** (that would have been redundant), and no **long long long long int**. A **long long** is guaranteed to hold at least 64 bits.

The *floating-point types* are **float**, **double**, and **long double**. They are C++'s approximation of real numbers.

The *integer types* (sometimes called *integral types*) are **bool**, **char**, **short**, **int**, **long**, and **long long** and their unsigned variants. Note that an enumeration type or value can often be used where an integer type or value is needed.

The sizes of built-in types are discussed in §3.8, §17.3.1, and §25.5.1. Pointers and arrays are discussed in Chapters 17 and 18. References are discussed in §8.5.4–6.

### A.8.1 Pointers

A *pointer* is an address of an object or a function. Pointers are stored in variables of pointer types. A valid object pointer holds the address of an object:

```
int x = 7;
int* pi = &x;           // pi points to x
int xx = *pi;          // *pi is the value of the object pointed to by pi, that is, 7
```

An invalid pointer is a pointer that does not hold the value of an object:

```
int* pi2;              // uninitialized
*pi2 = 7;              // undefined behavior
pi2 = nullptr;          // the null pointer (pi2 is still invalid)
*pi2 = 7;              // undefined behavior

pi2 = new int(7);       // now pi2 is valid
int xxx = *pi2;         // fine: xxx becomes 7
```

We try to have invalid pointers hold the null pointer (**nullptr**) so that we can test it:

```
if (p2 == nullptr) {   // "if invalid"
    // don't use *p2
}
```

Or simply

```
if (p2) {             // "if valid"
    // use *p2
}
```

See §17.4 and §18.6.4.

The operations on a (non-**void**) object pointer are:

Pointer operations	
<b>*p</b>	dereference/indirection
<b>p[i]</b>	dereference/subscripting
<b>p=q</b>	assignment and initialization
<b>p==q</b>	equality
<b>p!=q</b>	inequality
<b>p+i</b>	add integer
<b>p-i</b>	subtract integer
<b>p-q</b>	distance: subtract pointers
<b>++p</b>	pre-increment (move forward)
<b>p++</b>	post-increment (move forward)
<b>--p</b>	pre-decrement (move backward)
<b>p--</b>	post-decrement (move backward)
<b>p+=i</b>	move forward <b>i</b> elements
<b>p-=i</b>	move backward <b>i</b> elements

Note that any form of pointer arithmetic (e.g., **++p** and **p+=7**) is allowed only for pointers into an array and that the effect of dereferencing a pointer pointing outside the array is undefined (and most likely not checked by the compiler or the language run-time system). The comparisons **<**, **<=**, **>**, and **>=** can also be used for pointers of the same type into the same object or array.

The only operations on a **void\*** pointer are copying (assignment or initialization), casting (type conversion), and comparison (**==**, **!=**, **<**, **<=**, **>**, and **>=**).

A pointer to function (§27.2.5) can only be copied and called. For example:

```
using Handle_type = void (*)(int);
void my_handler(int);
Handle_type handle = my_handler;
handle(10);      // equivalent to my_handler(10)
```

## A.8.2 Arrays

An *array* is a fixed-length contiguous sequence of objects (elements) of a given type:

```
int a[10];           // 10 ints
```

If an array is global, its elements will be initialized to the appropriate default value for the type. For example, the value of `a[7]` will be `0`. If the array is local (a variable declared in a function) or allocated using `new`, elements of built-in types will be uninitialized and elements of class types will be initialized as required by the class's constructors.

The name of an array is implicitly converted to a pointer to its first element. For example:

```
int* p = a;    // p points to a[0]
```

An array or a pointer to an element of an array can be subscripted using the `[]` operator. For example:

```
a[7] = 9;
int xx = p[6];
```

Array elements are numbered starting with 0; see §18.6.

Arrays are not range checked, and since they are often passed as pointers, the information to range check them is not reliably available to users. Prefer `vector`.

The size of an array is the sum of the sizes of its elements. For example:

```
int a[max];    // sizeof(a); that is, sizeof(int)*max
```

You can define and use an array of an array (a two-dimensional array), an array of an array of an array, etc. (multidimensional arrays). For example:

```
double da[100][200][300];    // 300 elements of type
                                // 200 elements of type
                                // 100 type double
da[7][9][11] = 0;
```

Nontrivial uses of multidimensional arrays are subtle and error-prone; see §24.4. If you have a choice, prefer a `Matrix` library (such as the one in Chapter 24).

### A.8.3 References

A *reference* is an alias (alternative name) for an object:

```
int a = 7;
int& r = a;
r = 8; // a becomes 8
```

References are most common as function parameters, where they are used to avoid copying:

```
void f(const string& s);
// ...
f("this string could be somewhat costly to copy, so we use a reference");
```

See §8.5.4–6.

## A.9 Functions

A *function* is a named piece of code taking a (possibly empty) set of arguments and optionally returning a value. A function is declared by giving the return type followed by its name followed by the parameter list:

```
char f(string, int);
```

So, **f** is a function taking a **string** and an **int** returning a **char**. If the function is just being declared, the declaration is terminated by a semicolon. If the function is being defined, the argument declaration is followed by the function body:

```
char f(string s, int i) { return s[i]; }
```

The function body must be a block (§8.2) or a **try**-block (§5.6.3).

A function declared to return a value must return a value (using the **return**-statement):

```
char f(string s, int i) { char c = s[i]; } // error: no value returned
```

The **main()** function is the odd exception to that rule (§A.1.2). Except for **main()**, if you don't want to return a value, declare the function **void**; that is, use **void** as the “return type”:

```
void increment(int& x) { ++x; } // OK: no return value required
```

A function is called using the call operator (application operator), **( )**, with an acceptable list of arguments:

```
char x1 = f(1,2); // error: f()'s first argument must be a string
string s = "Battle of Hastings";
char x2 = f(s); // error: f() requires two arguments
char x3 = f(s,2); // OK
```

For more information about functions, see Chapter 8.

A function definition can be prefixed with `constexpr`. In that case, it must be simple enough for the compiler to evaluate when called with constant expression arguments. A `constexpr` function can be used in a constant expression (§8.5.9).

### A.9.1 Overload resolution

*Overload resolution* is the process of choosing a function to call based on a set of arguments. For example:

```
void print(int);
void print(double);
void print(const std::string&);

print(123);    // use print(int)
print(1.23);   // use print(double)
print("123");  // use print(const string&)
```

It is the compiler's job to pick the right function according to the language rules. Unfortunately, in order to cope with complicated examples, the language rules are quite complicated. Here we present a simplified version.

Finding the right version to call from a set of overloaded functions is done by looking for a best match between the type of the argument expressions and the parameters (formal arguments) of the functions. To approximate our notions of what is reasonable, a series of criteria is tried in order:

1. Exact match, that is, match using no or only trivial conversions (for example, array name to pointer, function name to pointer to function, and `T` to `const T`)
2. Match using promotions, that is, integral promotions (`bool` to `int`, `char` to `int`, `short` to `int`, and their unsigned counterparts; see §A.8) and `float` to `double`
3. Match using standard conversions, for example, `int` to `double`, `double` to `int`, `double` to `long double`, `Derived*` to `Base*` (§14.3), `T*` to `void*` (§17.8), `int` to `unsigned int` (§25.5.3)
4. Match using user-defined conversions (§A.5.2.3)
5. Match using the ellipsis `...` in a function declaration (§A.9.3)

If two matches are found at the highest level where a match is found, the call is rejected as ambiguous. The resolution rules are thus elaborate primarily to take into account the elaborate rules for built-in numeric types (§A.5.3).

For overload resolution based on multiple arguments, we first find the best match for each argument. If one function is at least as good a match as all other functions for every argument and is a better match than all other functions for one argument, that function is chosen; otherwise the call is ambiguous. For example:

```
void f(int, const string&, double);
void f(int, const char*, int);

f(1,"hello",1);           // OK: call f(int, const char*, int)
f(1,string("hello"),1.0); // OK: call f(int, const string&, double)
f(1, "hello",1.0);       // error: ambiguous
```

In the last call, the `"hello"` matches `const char*` without a conversion and `const string&` only with a conversion. On the other hand, `1.0` matches `double` without a conversion, but `int` only with a conversion, so neither `f()` is a better match than the other.

If these simplified rules don't agree with what your compiler says and what you thought reasonable, please first consider if your code is more complicated than necessary. If so, simplify your code; if not, consult an expert-level reference.

### A.9.2 Default arguments

A general function sometimes needs more arguments than are needed for the most common cases. To handle that, a programmer may provide default arguments to be used if a caller of a function doesn't specify an argument. For example:

```
void f(int, int=0, int=0);
f(1,2,3);
f(1,2);   // calls f(1,2,0)
f(1);     // calls f(1,0,0)
```

Only trailing arguments can be defaulted and left out in a call. For example:

```
void g(int, int =7, int);    // error: default for non-trailing argument
f(1,1);                      // error: second argument missing
```

Overloading can be an alternative to using default arguments (and vice versa).

### A.9.3 Unspecified arguments

It is possible to specify a function without specifying the number or types of its arguments. This is indicated by an ellipsis (`...`), meaning “and possibly more

arguments.” For example, here is the declaration of and some calls to what is arguably the most famous C function, **printf()** (§27.6.1, §B.11.2):

```
void printf(const char* format ...); // takes a format string and maybe more

int x = 'x';
printf("hello, world!");
printf("print a char '%c'\n",x); // print the int x as a char
printf("print a string \"%s\"",x); // shoot yourself in the foot
```

The “format specifiers” in the format string, such as **%c** and **%s**, determine if and how further arguments are used. As demonstrated, this can lead to nasty type errors. In C++, unspecified arguments are best avoided.

#### A.9.4 Linkage specifications

C++ code is often used in the same program as C code; that is, parts of a program are written in C++ (and compiled by a C++ compiler) and other parts in C (and compiled by a C compiler). To ease that, C++ offers *linkage specifications* for the programmer to say that a function obeys C linkage conventions. A C linkage specification can be placed in front of a function declaration:

```
extern "C" void callable_from_C(int);
```

Alternatively it can apply to all declarations in a block:

```
extern "C" {
    void callable_from_C(int);
    int and_this_one_also(double, int*);
    /* . . . */
}
```

For details of use, see §27.2.3.

C doesn’t offer function overloading, so you can put a C linkage specification on at most one version of a C++ overloaded function.

#### A.10 User-defined types

There are two ways for a programmer to define a new (user-defined) type: as a class (**class**, **struct**, or **union**; see §A.12) and as an enumeration (**enum**; see §A.11).

### A.10.1 Operator overloading

A programmer can define the meaning of most operators to take operands of one or more user-defined types. It is not possible to change the standard meaning of an operator for built-in types or to introduce a new operator. The name of a user-defined operator (“overloaded operator”) is the operator prefixed by the keyword **operator**; for example, the name of a function defining **+** is **operator +**:

```
Matrix operator+(const Matrix&, const Matrix&);
```

For examples, see **std::ostream** (Chapters 10–11), **std::vector** (Chapters 17–19, §B.4), **std::complex** (§B.9.3), and **Matrix** (Chapter 24).

All but the following operators can be user-defined:

```
?:: . .* :: sizeof typeid alignas noexcept
```

Functions defining the following operators must be members of a class:

```
= [] () ->
```

All other operators can be defined as member functions or as freestanding functions.

Note that every user-defined type has **=** (assignment and initialization), **&** (address of), and **,** (comma) defined by default.

Be restrained and conventional with operator overloading.

## A.11 Enumerations

An *enumeration* defines a type with a set of named values (*enumerators*):

```
enum Color { green, yellow, red }; // "plain" enumeration
enum class Traffic_light { yellow, red, green }; // scoped enumeration
```

The enumerators of an **enum class** are in the scope of the enumeration, whereas the enumerators of a “plain” **enum** are exported to the scope of the **enum**. For example:

```
Color col = red; // OK
Traffic_light tl = red; // error: cannot convert integer value
// (i.e., Color::red) to Traffic_light
```

By default the value of the first enumerator is **0**, so that **Color::green==0**, and the values increase by one, so that **Color's yellow==1** and **red==2**. It is also possible to explicitly define the value of an enumerator:

```
enum Day { Monday=1, Tuesday, Wednesday };
```

Here, we get **Monday==1**, **Tuesday==2**, and **Wednesday==3**.

Enumerators and enumeration values of a “plain” **enum** implicitly convert to integers, but integers do not implicitly convert to enumeration types:

```
int x = green;           // OK: implicit Color-to-int conversion
Color c = green;         // OK
c = 2;                  // error: no implicit int-to-Color conversion
c = Color(2);            // OK: (unchecked) explicit conversion
int y = c;                // OK: implicit Color-to-int conversion
```

Enumerators and enumeration values of an **enum class** do not convert to integers, and integers do not implicitly convert to enumeration types:

```
int x = Traffic_light::green; // error: no implicit Traffic_light-to-int conversion
Traffic_light c = green;      // error: no implicit int-to-Traffic_light conversion
```

For a discussion of the uses of enumerations, see §9.5.

## A.12 Classes

A *class* is a type for which the user defines the representation of its objects and the operations allowed on those objects:

```
class X {
public:
    // user interface
private:
    // implementation
};
```

A variable, function, or type defined within a class declaration is called a *member* of the class. See Chapter 9 for class technicalities.

### A.12.1 Member access

A **public** member can be accessed by users; a **private** member can be accessed only by the class’s own members:

```

class Date {
public:
    // ...
    int next_day();
private:
    int y, m, d;
};

void Date::next_day() { return d+1; }      // OK

void f(Date d)
{
    int nd = d.d+1;          // error: Date::d is private
    // ...
}

```

A **struct** is a class where members are by default **public**:

```

struct S {
    // members (public unless explicitly declared private)
};

```

For more details of member access, including a discussion of **protected**, see §14.3.4.

Members of an object can be accessed through a variable or referenced using the **.** (dot) operator or through a pointer using the **->** (arrow) operator:

```

struct Date {
    int d, m, y;
    int day() const { return d; }      // defined in-class
    int month() const;               // just declared; defined elsewhere
    int year() const;                // just declared; defined elsewhere
};

Date x;                                // access through variable
x.d = 15;                             // call through variable
int y = x.day();
Date* p = &x;
p->m = 7;                            // access through pointer
int z = p->month();                   // call through pointer

```

Members of a class can be referred to using the **::** (scope resolution) operator:

```
int Date::year() const { return y; } // out-of-class definition
```

Within a member function, we can refer to other members by their unqualified name:

```
struct Date {
    int d, m, y;
    int day() const { return d; }
    // ...
};
```

Such unqualified names refer to the member of the object for which the member function was called:

```
void f(Date d1, Date d2)
{
    d1.day();      // will access d1.d
    d2.day();      // will access d2.d
    // ...
}
```

### A.12.1.1 The **this** pointer

If we want to be explicit when referring to the object for which the member function is called, we can use the predefined pointer **this**:

```
struct Date {
    int d, m, y;
    int month() const { return this->m; }
    // ...
};
```

A member function declared **const** (a **const** member function) cannot modify the value of a member of the object for which it is called:

```
struct Date {
    int d, m, y;
    int month() const { ++m; } // error: month() is const
    // ...
};
```

For more information about **const** member functions, see §9.7.4.

### A.12.1.2 Friends

A function that is not a member of a class can be granted access to all members through a **friend** declaration. For example:

```
// needs access to Matrix and Vector members:
Vector operator*(const Matrix&, const Vector&);

class Vector {
    friend
    Vector operator*(const Matrix&, const Vector&); // grant access
    // ...
};

class Matrix {
    friend
    Vector operator*(const Matrix&, const Vector&); // grant access
    // ...
};
```

As shown, this is usually done for functions that need to access two classes. Another use of **friend** is to provide an access function that should not be called using the member access syntax. For example:

```
class Iter {
public:
    int distance_to(const iter& a) const;
    friend int difference(const Iter& a, const Iter& b);
    // ...
};

void f(Iter& p, Iter& q)
{
    int x = p.distance_to(q); // invoke using member syntax
    int y = difference(p,q); // invoke using "mathematical syntax"
    // ...
}
```

Note that a function declared **friend** cannot also be declared **virtual**.

### A.12.2 Class member definitions

Class members that are integer constants, functions, or types can be defined/initialized either *in-class* (§9.7.3) or *out-of-class* (§9.4.4):

```
struct S {
    int c = 1;
    int c2;

    void f() {}
    void f2();

    struct SS { int a; };
    struct SS2;
};
```

The members that were not defined in-class must be defined “elsewhere”:

```
int S::c2 = 7;

void S::f2() {}

struct S::SS2 { int m; };
```

If you want to initialize a data member with a value specified by the creator of an object, do it in a constructor.

Function members do not occupy space in an object:

```
struct S {
    int m;
    void f();
};
```

Here, **sizeof(S)==sizeof(int)**. That’s not actually guaranteed by the standard, but it is true for all implementations we know of. But note that a class with a virtual function has one “hidden” member to allow virtual calls (§14.3.1).

### A.12.3 Construction, destruction, and copy

You can define the meaning of initialization for an object of a class by defining one or more *constructors*. A constructor is a member function with the same name as its class and no return type:

```

class Date {
public:
    Date(int yy, int mm, int dd) :y{yy}, m{mm}, d{dd} {}
    // ...
private:
    int y,m,d;
};

Date d1 {2006,11,15};      // OK: initialization done by the constructor
Date d2;                  // error: no initializers
Date d3 {11,15};           // error: bad initializers (three initializers required)

```

Note that data members can be initialized by using an initializer list in the constructor (a base and member initializer list). Members will be initialized in the order in which they are declared in the class.

Constructors are typically used to establish a class's invariant and to acquire resources (§9.4.2–3).

Class objects are constructed “from the bottom up,” starting with base class objects (§14.3.1) in declaration order, followed by members in declaration order, followed by the code in the constructor itself. Unless the programmer does something really strange, this ensures that every object is constructed before use.

Unless declared **explicit**, a single-argument constructor defines an implicit conversion from its argument type to its class:

```

class Date {
public:
    Date(const char*);
    explicit Date(long);          // use an integer encoding of Date
    // ...
};

void f(Date);

Date d1 = "June 5, 1848";      // OK
f("June 5, 1848");             // OK

Date d2 = 2007*12*31+6*31+5;   // error: Date(long) is explicit
f(2007*12*31+6*31+5);         // error: Date(long) is explicit

Date d3(2007*12*31+6*31+5);   // OK

```

```
Date d4 = Date{2007*12*31+6*31+5};           // OK
f(Date{2007*12*31+6*31+5});                 // OK
```

Unless a class has bases or members that require explicit arguments, and unless the class has other constructors, a default constructor is automatically generated. This default constructor initializes each base or member that has a default constructor (leaving members without default constructors uninitialized). For example:

```
struct S {
    string name, address;
    int x;
};
```

This **S** has an implicit constructor **S{}** that initializes **name** and **address**, but not **x**. In addition, a class without a constructor can be initialized using an initializer list:

```
S s1 {"Hello!"};           // s1 becomes { "Hello! ",0}
S s2 {"Howdy!", 3};
S* p = new S{"G'day!"}; // *p becomes {"G'day",0};
```

As shown, trailing unspecified values become the default value (here, **0** for the **int**).

### A.12.3.1 Destructors

You can define the meaning of an object being destroyed (e.g., going out of scope) by defining a *destructor*. The name of a destructor is **~** (the complement operator) followed by the class name:

```
class Vector { // vector of doubles
public:
    explicit Vector(int s) : sz{s}, p{new double[s]} {} // constructor
    ~Vector() { delete[] p; }                         // destructor
    // ...
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(s);
    // ...
} // v will be destroyed upon exit from f(); Vector's destructor will be called for v
```

Destructors that invoke the destructors of members of a class can be generated by the compiler, and if a class is to be used as a base class, it usually needs a **virtual** destructor; see §17.5.2.

A destructor is typically used to “clean up” and release resources.

Class objects are destructed “from the top down” starting with the code in the destructor itself, followed by members in declaration order, followed by the base class objects in declaration order, that is, in reverse order of construction.

### A.12.3.2 Copying

You can define the meaning of *copying* an object of a class:

```
class Vector { // vector of doubles
public:
    explicit Vector(int s) : sz{s}, p{new double[s]} { } // constructor
    ~Vector() { delete[] p; } // destructor
    Vector(const Vector&); // copy constructor
    Vector& operator=(const Vector&); // copy assignment
    // ...
private:
    int sz;
    double* p;
};

void f(int ss)
{
    Vector v(ss);
    Vector v2 = v; // use copy constructor
    // ...
    v = v2; // use copy assignment
    // ...
}
```

By default (that is, unless you define a copy constructor and a copy assignment), the compiler will generate copy operations for you. The default meaning of copy is memberwise copy; see also §14.2.4 and §18.3.

### A.12.3.3 Moving

You can define the meaning of *moving* an object of a class:

```
class Vector { // vector of doubles
public:
    explicit Vector(int s) : sz{s}, p{new double[s]} { } // constructor
    ~Vector() { delete[] p; } // destructor
```

```

Vector(Vector&&);           // move constructor
Vector& operator=(Vector&&); // move assignment
// ...
private:
    int sz;
    double* p;
};

Vector f(int ss)
{
    Vector v(ss);
    // ...
    return v;      // use move constructor
}

```

By default (that is, unless you define a copy constructor and a copy assignment), the compiler will generate move operations for you. The default meaning of move is memberwise move; see also §18.3.4.

#### A.12.4 Derived classes

A class can be defined as derived from other classes, in which case it inherits the members of the classes from which it is derived (its base classes):

```

struct B {
    int mb;
    void fb() {};
};

class D : B {
    int md;
    void fd();
};

```

Here **B** has two members, **mb** and **fb()**, whereas **D** has four members, **mb**, **fb()**, **md**, and **fd()**.

Like members, bases can be **public** or **private**:

```

Class DD : public B1, private B2 {
    // ...
};

```

So, the **public** members of **B1** become **public** members of **DD**, whereas the **public** members of **B2** become **private** members of **DD**. A derived class has no special

access to members of its bases, so **DD** does not have access to the **private** members of **B1** or **B2**.

A class with more than one direct base class (such as **DD**) is said to use *multiple inheritance*.

A pointer to a derived class, **D**, can be implicitly converted to a pointer to its base class, **B**, provided **B** is accessible and is unambiguous in **D**. For example:

```
struct B { };
struct B1: B { }; // B is a public base of B1
struct B2: B { }; // B is a public base of B2
struct C { };
struct DD : B1, B2, private C { };

DD* p = new DD;
B1* pb1 = p; // OK
B* pb = p; // error: ambiguous: B1::B or B2::B
C* pc = p; // error: DD::C is private
```

Similarly, a reference to a derived class can be implicitly converted to an unambiguous and accessible base class.

For more information about derived classes, see §14.3. For more information about **protected**, see an expert-level textbook or reference.

#### A.12.4.1 Virtual functions

A *virtual function* is a member function that defines a calling interface to functions of the same name taking the same argument types in derived classes. When calling a virtual function, the function invoked by the call will be the one defined for the most derived class. The derived class is said to override the virtual function in the base class.

```
class Shape {
public:
    virtual void draw(); // virtual means "can be overridden"
    virtual ~Shape() {} // virtual destructor
    // ...
};

class Circle : public Shape {
public:
    void draw(); // override Shape::draw
    ~Circle(); // override Shape::~Shape()
    // ...
};
```

Basically, the virtual functions of a base class (here, **Shape**) define a calling interface for the derived class (here, **Circle**):

```
void f(Shape& s)
{
    // ...
    s.draw();
}

void g()
{
    Circle c{Point{0,0}, 4};
    f(c);      // will call Circle's draw
}
```

Note that **f()** doesn't know about **Circles**, only about **Shapes**. An object of a class with a virtual function contains one extra pointer to allow it to find the set of overriding functions; see §14.3.

Note that a class with virtual functions usually needs a virtual destructor (as **Shape** has); see §17.5.2.

The wish to override a base class's virtual function can be made explicit using the **override** suffix. For example:

```
class Square : public Shape {
public:
    void draw() override; // override Shape::draw
    ~Circle() override; // override Shape::~Shape()
    void silly() override; // error: Shape does not have a virtual Shape::silly()
    // ...
};
```

#### A.12.4.2 Abstract classes

An *abstract class* is a class that can be used only as a base class. You cannot make an object of an abstract class:

```
Shape s;           // error: Shape is abstract
```

```
class Circle : public Shape {
```

```
public:
    void draw();      // override Shape::draw
    // ...
};

Circle c{p,20};           // OK: Circle is not abstract
```

The most common way of making a class abstract is to define at least one pure virtual function. A *pure virtual function* is a virtual function that requires overriding:

```
class Shape {
public:
    virtual void draw() = 0;      // =0 means "pure"
    // ...
};
```

See §14.3.5.

The rarer, but equally effective, way of making a class abstract is to declare all its constructors **protected** (§14.2.1).

#### A.12.4.3 Generated operations

When you define a class, it will by default have several operations defined for its objects:

- Default constructor
- Copy operations (copy assignment and copy initialization)
- Move operations (move assignment and move initialization)
- Destructor

Each is (again by default) defined to apply recursively to each of its base classes and members. Construction is done “bottom-up,” that is, bases before members. Destruction is done “top-down,” that is, members before bases. Members and bases are constructed in order of appearance and destroyed in the opposite order. That way, constructor and destructor code always relies on well-defined base and member objects. For example:

```
struct D : B1, B2 {
    M1 m1;
    M2 m2;
};
```

Assuming that **B1**, **B2**, **M1**, and **M2** are defined, we can now write

```
D f()
{
    D d;           // default initialization
    D d2 = d;     // copy initialization
    d = D{};     // default initialization followed by copy assignment
    return d;     // d is moved out of f()
} // d and d2 are destroyed here
```

For example, the default initialization of **d** invokes four default constructors (in order): **B1::B1()**, **B2::B2()**, **M1::M1()**, and **M2::M2()**. If one of those doesn't exist or can't be called, the construction of **d** fails. At the **return**, four move constructors are invoked (in order): **B1::B1()**, **B2::B2()**, **M1::M1()**, and **M2::M2()**. If one of those doesn't exist or can't be called, the **return** fails. The destruction of **d** invokes four destructors (in order): **M2::~M2()**, **M1::~M1()**, **B2::~B2()**, and **B1::~B1()**. If one of those doesn't exist or can't be called, the destruction of **d** fails. Each of these constructors and destructors can be either user-defined or generated.

The implicit (compiler-generated) default constructor is not defined (generated) if a class has a user-defined constructor.

### A.12.5 Bitfields

A *bitfield* is a mechanism for packing many small values into a word or to match an externally imposed bit-layout format (such as a device register). For example:

```
struct PPN { // R6000 Physical Page Number
    unsigned int PFN : 22;      // Page Frame Number
    int : 3;                  // unused
    unsigned int CCA : 3;     // Cache Coherency Algorithm
    bool nonreachable : 1;
    bool dirty : 1;
    bool valid : 1;
    bool global : 1;
};
```

Packing the bitfields into a word left to right leads to a layout of bits in a word like this (see §25.5.5):

position:	31:	8:	5:	2: 1: 0:
PPN:	22	3	3	1 1 1
name:	PFN	unused	CCA	dirty global valid

A bitfield need not have a name, but if it doesn't, you can't access it.

Surprisingly, packing many small values into a single word does not necessarily save space. In fact, using one of those values often wastes space compared to using a `char` or an `int` to represent even a single bit. The reason is that it takes several instructions (which have to be stored in memory somewhere) to extract a bit from a word and to write a single bit of a word without modifying other bits of a word. Don't try to use bitfields to save space unless you need lots of objects with tiny data fields.

### A.12.6 Unions

A *union* is a class where all members are allocated starting at the same address. A union can hold only one element at a time, and when a member is read it must be the same as was last written. For example:

```
union U {
    int x;
    double d;
}

U a;
a.x = 7;
int x1 = a.x; // OK
a.d = 7.7;
int x2 = a.x; // oops
```

The rule requiring consistent reads and writes is not checked by the compiler. You have been warned.

## A.13 Templates

A *template* is a class or a function parameterized by a set of types and/or integers:

```
template<typename T>
class vector {
public:
    // ...
    int size() const;
private:
    int sz;
    T* p;
};
```

```
template<class T>
int vector<T>::size() const
{
    return sz;
}
```

In a template argument list, **class** means type; **typename** is an equivalent alternative. A member function of a template class is implicitly a template function with the same template arguments as its class.

Integer template arguments must be constant expressions:

```
template<typename T, int sz>
class Fixed_array {
public:
    T a[sz];
    // ...
    int size() const { return sz; };
};

Fixed_array<char,256> x1; // OK
int var = 226;
Fixed_array<char,var> x2; // error: non-const template argument
```

### A.13.1 Template arguments

Arguments for a template class are specified whenever its name is used:

<b>vector&lt;int&gt; v1;</b>	<i>// OK</i>
<b>vector v2;</b>	<i>// error: template argument missing</i>
<b>vector&lt;int,2&gt; v3;</b>	<i>// error: too many template arguments</i>
<b>vector&lt;2&gt; v4;</b>	<i>// error: type template argument expected</i>

Arguments for template functions are typically deduced from the function arguments:

```
template<class T>
T find(vector<T>& v, int i)
{
    return v[i];
}

vector<int> v1;
vector<double> v2;
```

```
// ...
int x1 = find(v1,2);           // find()'s T is int
int x2 = find(v2,2);           // find()'s T is double
```

It is possible to define a template function for which it is not possible to deduce its template arguments from its function arguments. In that case we must specify the missing template arguments explicitly (exactly as for class templates). For example:

```
template<class T, class U> T* make(const U& u) { return new T{u}; }
int* pi = make<int>(2);
Node* pn = make<Node>(make_pair("hello",17));
```

This works if a **Node** can be initialized by a **pair<const char \*,int>** (§B.6.3). Only trailing template arguments can be left out of an explicit argument specialization (to be deduced).

### A.13.2 Template instantiation

A version of a template for a specific set of template arguments is called a *specialization*. The process of generating specializations from a template and a set of arguments is called *template instantiation*. Usually, the compiler generates a specialization from a template and a set of template arguments, but the programmer can also define a specific specialization. This is usually done when a general template is unsuitable for a particular set of arguments. For example:

```
template<class T> struct Compare {           // general compare
    bool operator()(const T& a, const T& b) const
    {
        return a < b;
    }
};

template<> struct Compare<const char*> {      // compare C-style strings
    bool operator()(const char* a, const char* b) const
    {
        return strcmp(a,b)==0;
    }
};

Compare<int> c2;                                // general compare
Compare<const char*> c;                         // C-style string compare
```

```
bool b1 = c2(1,2);           // use general compare
bool b2 = c("asd","dfg");    // use C-style string compare
```

For functions, the rough equivalent is achieved through overloading:

```
template<class T> bool compare(const T& a, const T& b)
{
    return a<b;
}

bool compare (const char* a, const char* b) // compare C-style strings
{
    return strcmp(a,b)==0;
}

bool b3 = compare(2,3);           // use general compare
bool b4 = compare("asd","dfg");    // use C-style string compare
```

Separate compilation of templates (i.e., keeping declarations only in header files and unique definitions in .cpp files) does not work portably, so if a template needs to be used in several .cpp files, put its complete definition in a header file.

### A.13.3 Template member types

A template can have members that are types and members that are not types (such as data members and member functions). This means that in general, it can be hard to tell whether a member name refers to a type or to a non-type. For language-technical reasons, the compiler has to know, so occasionally we must tell it. For that, we use the keyword **typename**. For example:

```
template<class T> struct Vec {
    typedef T value_type;      // a member type
    static int count;          // a data member
    // ...
};

template<class T> void my_fct(Vec<T>& v)
{
    int x = Vec<T>::count;    // by default member names
                                // are assumed to refer to non-types
    v.count = 7;              // a simpler way to refer to a non-type member
    typename Vec<T>::value_type xx = x; // typename is needed here
    // ...
}
```

For more information about templates, see Chapter 19.

## A.14 Exceptions

An exception is used (with a `throw` statement) to tell a caller about an error that cannot be handled locally. For example, move `Bad_size` out of `Vector`:

```
struct Bad_size {
    int sz;
    Bad_size(int s) : sz{s} {}
};

class Vector {
    Vector(int s) { if (s<0 || maxsize<s) throw Bad_size{s}; }
    // ...
};
```

Usually, we throw a type that is defined specifically to represent a particular error. A caller can catch an exception:

```
void f(int x)
{
    try {
        Vector v(x); // may throw
        // ...
    }
    catch (Bad_size bs) {
        cerr << "Vector with bad size (" << bs.sz << ")\n";
        // ...
    }
}
```

A “catch all” clause can be used to catch every exception:

```
try {
    // ...
} catch (...) { // catch all exceptions
    // ...
}
```

Usually, the RAII (“Resource Acquisition Is Initialization”) technique is better (simpler, easier, more reliable) than using lots of explicit `trys` and `catches`; see §19.5.

A **throw** without an argument (i.e., **throw;**) re-throws the current exception. For example:

```
try {
    // ...
} catch (Some_exception& e) {
    // do local cleanup
    throw;      // let my caller do the rest
}
```

You can define your own types for use as exceptions. The standard library defines a few exception types that you can also use; see §B.2.1. Never use a built-in type as an exception (someone else might have done that and your exceptions might be confused with those).

When an exception is thrown, the run-time support system for C++ searches “up the call stack” for a **catch**-clause with a type that matches the type of the object thrown; that is, it looks through **try**-statements in the function that threw, then through the function that called the function that threw, then through the function that called the function that called, etc., until it finds a match. If it doesn’t find a match, the program terminates. In each function encountered in this search of a matching **catch**-clause and in each scope on the way, destructors are called to clean up. This process is called *stack unwinding*.

An object is considered constructed once its constructor has completed and will then be destroyed during unwinding or any other exit from its scope. This implies that partially constructed objects (with some members or bases constructed and some not), arrays, and variables in a scope are correctly handled. Sub-objects are destroyed if and only if they have been constructed.

Do not throw an exception so that it leaves a destructor. This implies that a destructor should not fail. For example:

```
X::~X() { if (in_a_real_mess()) throw Mess{}; }      // never do this!
```

The primary reason for this Draconian advice is that if a destructor throws (and doesn’t itself catch the exception) during unwinding, we wouldn’t know which exception to handle. It is worthwhile to go to great lengths to avoid a destructor exiting by a throw because we know of no systematic way of writing correct code where that can happen. In particular, no standard library facility is guaranteed to work if that happens.

## A.15 Namespaces

A *namespace* groups related declarations together and is used to prevent name clashes:

```
int a;

namespace Foo {
    int a;
    void f(int i)
    {
        a+= i;      // that's Foo's a (Foo::a)
    }
}

void f(int);

int main()
{
    a = 7;          // that's the global a (::a)
    f(2);          // that's the global f (::f)
    Foo::f(3);      // that's Foo's f
    ::f(4);          // that's the global f (::f)
}
```

Names can be explicitly qualified by their namespace name (e.g., **Foo::f(3)**) or by **::** (e.g., **::f(2)**), indicating the global scope.

All names from a namespace (here, the standard library namespace, **std**) can be made accessible by a single namespace directive:

```
using namespace std;
```

Be restrained in the use of **using** directives. The notational convenience offered by a **using** directive is achieved at the cost of potential name clashes. In particular, avoid **using** directives in header files. A single name from a namespace can be made available by a namespace declaration:

```
using Foo::g;
g(2);          // that's Foo's g (Foo::g)
```

For more information about namespaces, see §8.7.

## A.16 Aliases

We can define an *alias* for a name; that is, we can define a symbolic name that means exactly the same as what it refers to (for most uses of the name):

```
using Pint = int*;           // Pint means pointer to int

namespace Long_library_name { /* ... */ }
namespace Lib = Long_library_name; // Lib means Long_library_name

int x = 7;
int& r = x;           // r means x
```

A reference (§8.5.5, §A.8.3) is a run-time mechanism, referring to objects. The **using** (§20.5) and **namespace** aliases are compile-time mechanisms, referring to names. In particular, a **using** does not introduce a new type, just a new name for a type. For example:

```
using Pchar = char*;      // Pchar is a name for char*
Pchar p = "Idefix";       // OK: p is a char*
char* q = p;             // OK: p and q are both char*
int x = strlen(p);        // OK: p is a char*
```

Older code uses the keyword **typedef** (§27.3.1) rather than the (C++) **using** notation to define a type alias. For example:

```
typedef char* Pchar;      // Pchar is a name for char*
```

## A.17 Preprocessor directives

Every C++ implementation includes a *preprocessor*. In principle, the preprocessor runs before the compiler proper and transforms the source code we wrote into what the compiler sees. In reality, this action is integrated into the compiler and uninteresting except when it causes problems. Every line starting with **#** is a preprocessor directive.

### A.17.1 #include

We have used the preprocessor extensively to include headers. For example:

```
#include "file.h"
```

This is a directive that tells the preprocessor to include the contents of `file.h` at the point of the source text where the directive occurs. For standard headers, we can also use `<...>` instead of `"..."`. For example:

```
#include<vector>
```

That is the recommended notation for standard header inclusion.

### A.17.2 #define

The preprocessor implements a form of character manipulation called *macro substitution*. For example, we can define a name for a character string:

```
#define FOO bar
```

Now, whenever `FOO` is seen, `bar` will be substituted:

```
int FOO = 7;  
int FOOL = 9;
```

Given that, the compiler will see

```
int bar = 7;  
int FOOL = 9;
```

Note that the preprocessor knows enough about C++ names not to replace the `FOO` that's part of `FOOL`.

You can also define macros that take parameters:

```
#define MAX(x,y) (((x)>(y))?(x) : (y))
```

And we can use it like this:

```
int xx = MAX(FOO+1,7);  
int yy = MAX(++xx,9);
```

This will expand to

```
int xx = (((bar+1)>( 7))?(bar+1) : (7));  
int yy = (((++xx)>( 9))?(++xx) : (9));
```

Note how the parentheses were necessary to get the right result for `FOO+1`. Also note that `xx` was incremented twice in a very non-obvious way. Macros are

immensely popular – primarily because C programmers have few alternatives to using them. Common header files define thousands of macros. You have been warned!

If you must use macros, the convention is to name them using **ALL\_CAPITAL LETTERS**. No ordinary name should be in all capital letters. Don't depend on others to follow this sound advice. For example, we have found a macro called **max** in an otherwise reputable header file.

See also §27.8.



## Standard Library Summary

“All complexities should,  
if possible,  
be buried out of sight.”

—David J. Wheeler

This appendix summarizes key C++ standard library facilities. The summary is selective and geared to novices who want to get an overview of the standard library facilities and explore a bit beyond the sequence of topics in the book.

**B.1 Overview**

- B.1.1 Header files**
- B.1.2 Namespace `std`**
- B.1.3 Description style**

**B.2 Error handling**

- B.2.1 Exceptions**

**B.3 Iterators**

- B.3.1 Iterator model**
- B.3.2 Iterator categories**

**B.4 Containers**

- B.4.1 Overview**
- B.4.2 Member types**
- B.4.3 Constructors, destructors, and assignments**
- B.4.4 Iterators**
- B.4.5 Element access**
- B.4.6 Stack and queue operations**
- B.4.7 List operations**
- B.4.8 Size and capacity**
- B.4.9 Other operations**
- B.4.10 Associative container operations**

**B.5 Algorithms**

- B.5.1 Nonmodifying sequence algorithms**
- B.5.2 Modifying sequence algorithms**
- B.5.3 Utility algorithms**
- B.5.4 Sorting and searching**
- B.5.5 Set algorithms**
- B.5.6 Heaps**
- B.5.7 Permutations**
- B.5.8 `min` and `max`**

**B.6 STL utilities**

- B.6.1 Inserters**
- B.6.2 Function objects**
- B.6.3 `pair` and `tuple`**
- B.6.4 `initializer_list`**
- B.6.5 Resource management pointers**

**B.7 I/O streams**

- B.7.1 I/O streams hierarchy**
- B.7.2 Error handling**
- B.7.3 Input operations**
- B.7.4 Output operations**
- B.7.5 Formatting**
- B.7.6 Standard manipulators**

**B.8 String manipulation**

- B.8.1 Character classification**
- B.8.2 String**
- B.8.3 Regular expression matching**

**B.9 Numerics**

- B.9.1 Numerical limits**
- B.9.2 Standard mathematical functions**
- B.9.3 Complex**
- B.9.4 `valarray`**
- B.9.5 Generalized numerical algorithms**
- B.9.6 Random numbers**

**B.10 Time****B.11 C standard library functions**

- B.11.1 Files**
- B.11.2 The `printf()` family**
- B.11.3 C-style strings**
- B.11.4 Memory**
- B.11.5 Date and time**
- B.11.6 Etc.**

**B.12 Other libraries**

## B.1 Overview

This appendix is a reference. It is not intended to be read from beginning to end like a chapter. It (more or less) systematically describes key elements of the C++ standard library. It is not a complete reference, though; it is just a summary with a few key examples. Often, you will need to look at the chapters for a more complete explanation. Note also that this summary does not attempt to equal the precision and terminology of the standard. For more information, see Stroustrup, *The C++ Programming Language*. The complete definition is the ISO C++ standard,

but that document is not intended for or suitable for novices. Don't forget to use your online documentation.

What use is a selective (and therefore incomplete) summary? You can quickly look for a known operation or quickly scan a section to see what common operations are available. You may very well have to look elsewhere for a detailed explanation, but that's fine: now you have a clue as to what to look for. Also, this summary contains cross-references to tutorial material in the chapters. This appendix provides a compact overview of standard library facilities. Please do not try to memorize the information here; that's not what it is for. On the contrary, this appendix is a tool that can save you from spurious memorization.

This is a place to look for useful facilities – instead of trying to invent them yourself. Everything in the standard library (and especially everything featured in this appendix) has been useful to large groups of people. A standard library facility is almost certainly better designed, better implemented, better documented, and more portable than anything you could design and implement in a hurry. So when you can, prefer a standard library facility over “home brew.” Doing so will also make your code easier for others to understand.

If you are a sensible person, you'll find the sheer mass of facilities intimidating. Don't worry; ignore what you don't need. If you are a “details person,” you'll find much missing. However, completeness is what the expert-level guides and your online documentation offer. In either case, you'll find much that will seem mysterious, and possibly interesting. Explore some of it!

### B.1.1 Header files

The interfaces to standard library facilities are defined in headers. Use this section to gain an overview of what is available and to help guess where a facility might be defined and described:

The STL (containers, iterators, and algorithms)	
<code>&lt;algorithm&gt;</code>	algorithms; <code>sort()</code> , <code>find()</code> , etc. (§B.5, §21.1)
<code>&lt;array&gt;</code>	fixed-size array (§20.9)
<code>&lt;bitset&gt;</code>	array of <code>bool</code> (§25.5.2)
<code>&lt;deque&gt;</code>	double-ended queue
<code>&lt;functional&gt;</code>	function objects (§B.6.2)
<code>&lt;iterator&gt;</code>	iterators (§B.4.4)
<code>&lt;list&gt;</code>	doubly-linked list (§B.4, §20.4)
<code>&lt;forward_list&gt;</code>	singly-linked list
<code>&lt;map&gt;</code>	(key,value) <code>map</code> and <code>multimap</code> (§B.4, §21.6.1–3)

**The STL (containers, iterators, and algorithms) (*continued*)**

<b>&lt;memory&gt;</b>	allocators for containers
<b>&lt;queue&gt;</b>	<b>queue</b> and <b>priority_queue</b>
<b>&lt;set&gt;</b>	<b>set</b> and <b>multiset</b> (§B.4, §21.6.5)
<b>&lt;stack&gt;</b>	<b>stack</b>
<b>&lt;unordered_map&gt;</b>	hash maps (§21.6.4)
<b>&lt;unordered_set&gt;</b>	hash sets
<b>&lt;utility&gt;</b>	operators and <b>pair</b> (§B.6.3)
<b>&lt;vector&gt;</b>	<b>vector</b> (dynamically expandable) (§B.4, §20.8)

**I/O streams**

<b>&lt;iostream&gt;</b>	I/O stream objects (§B.7)
<b>&lt;fstream&gt;</b>	file streams (§B.7.1)
<b>&lt;sstream&gt;</b>	<b>string</b> streams (§B.7.1)
<b>&lt;iosfwd&gt;</b>	declare (but don't define) I/O stream facilities
<b>&lt;ios&gt;</b>	I/O stream base classes
<b>&lt;streambuf&gt;</b>	stream buffers
<b>&lt;istream&gt;</b>	input streams (§B.7)
<b>&lt;ostream&gt;</b>	output streams (§B.7)
<b>&lt;iomanip&gt;</b>	formatting and manipulators (§B.7.6)

**String manipulation**

<b>&lt;string&gt;</b>	<b>string</b> (§B.8.2)
<b>&lt;regex&gt;</b>	regular expressions (Chapter 23)

**Numerics**

<b>&lt;complex&gt;</b>	complex numbers and arithmetic (§B.9.3)
<b>&lt;random&gt;</b>	random number generation (§B.9.6)

**Numerics (*continued*)**

<code>&lt;valarray&gt;</code>	numeric arrays
<code>&lt;numeric&gt;</code>	generalized numeric algorithms, e.g., <code>accumulate()</code> (§B.9.5)
<code>&lt;limits&gt;</code>	numerical limits (§B.9.1)

**Utility and language support**

<code>&lt;exception&gt;</code>	exception types (§B.2.1)
<code>&lt;stdexcept&gt;</code>	exception hierarchy (§B.2.1)
<code>&lt;locale&gt;</code>	culture-specific formatting
<code>&lt;typeinfo&gt;</code>	standard type information (from <code>typeid</code> )
<code>&lt;new&gt;</code>	allocation and deallocation functions
<code>&lt;memory&gt;</code>	resource management pointers, e.g. <code>unique_ptr</code> (§B.6.5)

**Concurrency support**

<code>&lt;thread&gt;</code>	threads (beyond the scope of this book)
<code>&lt;future&gt;</code>	inter-thread communication (beyond the scope of this book)
<code>&lt;mutex&gt;</code>	mutual exclusion facilities (beyond the scope of this book)

**C standard libraries**

<code>&lt;cstring&gt;</code>	C-style string manipulation (§B.11.3)
<code>&lt;cstdio&gt;</code>	C-style I/O (§B.11.2)
<code>&lt;ctime&gt;</code>	<code>clock()</code> , <code>time()</code> , etc. (§B.11.5)
<code>&lt;cmath&gt;</code>	standard floating-point math functions (§B.9.2)
<code>&lt;cstdlib&gt;</code>	etc. functions: <code>abort()</code> , <code>abs()</code> , <code>malloc()</code> , <code>qsort()</code> , etc. (Chapter 27)
<code>&lt;cerrno&gt;</code>	C-style error handling (§24.8)
<code>&lt;cassert&gt;</code>	assert macro (§27.9)
<code>&lt;climits&gt;</code>	culture-specific formatting
<code>&lt;cfloat&gt;</code>	C-style floating-point limits (§B.9.1)

C standard libraries ( <i>continued</i> )	
<code>&lt;cstddef&gt;</code>	C language support; <code>size_t</code> , etc.
<code>&lt;cstdarg&gt;</code>	macros for variable argument processing
<code>&lt;csetjmp&gt;</code>	<code>setjmp()</code> and <code>longjmp()</code> (never use those)
<code>&lt;csignal&gt;</code>	signal handling
<code>&lt;cwchar&gt;</code>	wide characters
<code>&lt;cctype&gt;</code>	character type classification (§B.8.1)
<code>&lt;cwctype&gt;</code>	wide character type classification

For each of the C standard library headers, there is also a version without the initial `c` in its name and with a trailing `.h`, such as `<time.h>` for `<ctime>`. The `.h` versions define global names rather than names in namespace `std`.

Some – but not all – of the facilities defined in these headers are described in the sections below and in the chapters. If you need more information, look at your online documentation or an expert-level C++ book.

### B.1.2 Namespace std

The standard library facilities are defined in namespace `std`, so to use them, you need an explicit qualification, a `using` declaration, or a `using` directive:

```
std::string s;           // explicit qualification

using std::vector;      // using declaration
vector<int> v(7);

using namespace std;     // using directive
map<string,double> m;
```

In this book, we have used the `using` directive for `std`. Be very frugal with `using` directives; see §A.15.

### B.1.3 Description style

A full description of even a simple standard library operation, such as a constructor or an algorithm, can take pages. Consequently, we use an extremely abbreviated style of presentation. For example:

**Examples of notation**

**p=op(b,e,x)**    **op** does something to the range **[b:e)** and **x**, returning **p**.

**foo(x)**         **foo** does something to **x**, but returns no result.

**bar(b,e,x)**    Does **x** have something to do with **[b:e)**?

We try to be mnemonic in our choice of identifiers, so **b,e** will be iterators specifying a range, **p** a pointer or an iterator, and **x** some value, all depending on context. In this notation, only the commentary distinguishes no result from a Boolean result, so you can confuse those if you try hard enough. For an operation returning **bool**, the explanation usually ends with a question mark.

Where an algorithm follows the usual pattern of returning the end of an input sequence to indicate “failure,” “not found,” etc. (§B.3.1), we do not mention that explicitly.

## B.2 Error handling

The standard library consists of components developed over a period of over 40 years. Thus, their style and approaches to error handling are not consistent.

- C-style libraries consist of functions, many of which set **errno** to indicate that an error happened; see §24.8.
- Many algorithms operating on a sequence of elements return an iterator to the one-past-the-last element to indicate “not found” or “failure.”
- The I/O streams library relies on a state in each stream to reflect errors and may (if the user requests it) throw exceptions to indicate errors; see §10.6, §B.7.2.
- Some standard library components, such as **vector**, **string**, and **bitset**, throw exceptions to indicate errors.

The standard library is designed so that all facilities obey “the basic guarantee” (see §19.5.3); that is, even if an exception is thrown, no resource (such as memory) is leaked and no invariant for a standard library class is broken.

### B.2.1 Exceptions

Some standard library facilities report errors by throwing exceptions:

Standard library exceptions	
<code>bitset</code>	throws <code>invalid_argument</code> , <code>out_of_range</code> , <code>overflow_error</code>
<code>dynamic_cast</code>	throws <code>bad_cast</code> if it cannot perform a conversion
<code>iostream</code>	throws <code>ios_base::failure</code> if exceptions are enabled
<code>new</code>	throws <code>bad_alloc</code> if it cannot allocate memory
<code>regex</code>	throws <code>regex_error</code>
<code>string</code>	throws <code>length_error</code> , <code>out_of_range</code>
<code>typeid</code>	throws <code>bad_typeid</code> if it cannot deliver a <code>type_info</code>
<code>vector</code>	throws <code>out_of_range</code>

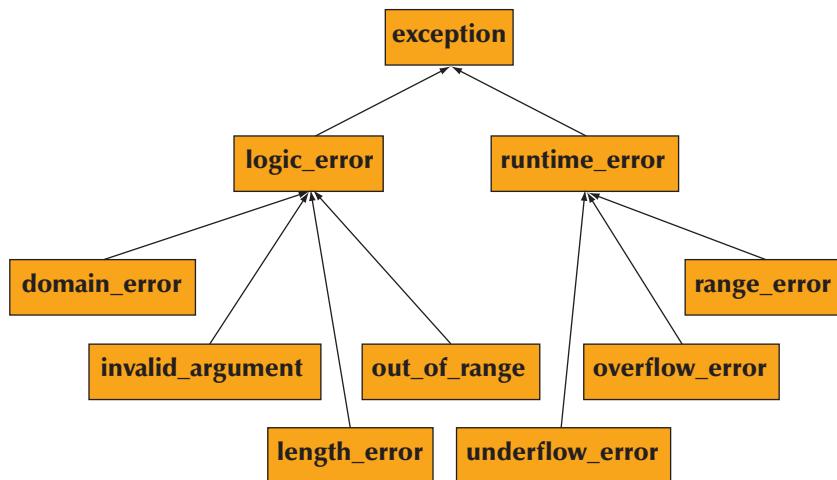
These exceptions may be encountered in any code that directly or indirectly uses these facilities. Unless you *know* that no facility is used in a way that could throw an exception, it is a good idea to always catch one of the root classes of the standard library exception hierarchy (such as `exception`) somewhere (e.g., in `main()`).

We strongly recommend that you do not throw built-in types, such as `ints` and C-style strings. Instead, throw objects of types specifically defined to be used as exceptions. A class derived from the standard library class `exception` can be used for that:

```
class exception {
public:
    exception();
    exception(const exception&);
    exception& operator=(const exception&);
    virtual ~exception();
    virtual const char* what() const;
};
```

The `what()` function can be used to obtain a string that is supposed to indicate something about the error that caused the exception.

This hierarchy of standard exception classes may help by providing a classification of exceptions:



You can define an exception by deriving from a standard library exception like this:

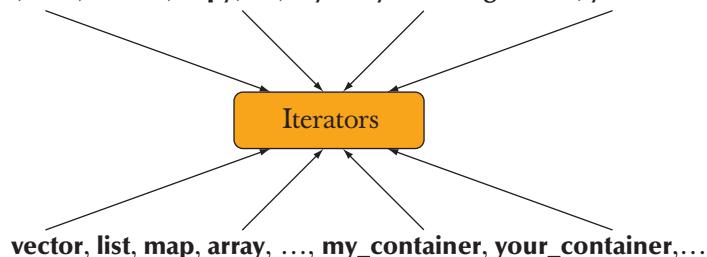
```

struct My_error : runtime_error {
    My_error(int x) : interesting_value{x} {}
    int interesting_value;
    const char* what() const override { return "My_error"; }
};
  
```

## B.3 Iterators

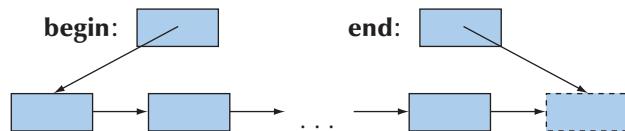
Iterators are the glue that ties standard library algorithms to their data. Conversely, you can say that iterators are the mechanism used to minimize an algorithm's dependence on the data structures on which it operates (§20.3):

`sort, find, search, copy, ..., my_very_own_algorithm, your_code, ...`



### B.3.1 Iterator model

An iterator is akin to a pointer in that it provides operations for indirect access (e.g., `*` for dereferencing) and for moving to a new element (e.g., `++` for moving to the next element). A sequence of elements is defined by a pair of iterators defining a half-open range `[begin:end]`:



That is, `begin` points to the first element of the sequence and `end` points to one beyond the last element of the sequence. Never read from or write to `*end`. Note that the empty sequence has `begin==end`; that is, `[p:p)` is the empty sequence for any iterator `p`.

To read a sequence, an algorithm usually takes a pair of iterators (`b,e`) and iterates using `++` until the end is reached:

```

while (b!=e) {   // use != rather than <
    // do something
    ++b;      // go to next element
}
    
```

Algorithms that search for something in a sequence usually return the end of the sequence to indicate “not found”; for example:

```

p = find(v.begin(),v.end(),x);      // look for x in v
if (p!=v.end()) {
    // x found at p
}
else {
    // x not found in [v.begin():v.end())
}
    
```

See §20.3.

Algorithms that write to a sequence often are given only an iterator to its first element. In that case, it is the programmer’s responsibility not to write beyond the end of that sequence. For example:

```

template<class Iter> void f(Iter p, int n)
{
    while (n>0) *p++ = --n;
}
    
```

```
vector<int> v(10);
f(v.begin(),v.size());           // OK
f(v.begin(),1000);              // big trouble
```

Some standard library implementations range check – that is, throw an exception – for that last call of `f()`, but you can't rely on that for portable code; many implementations don't check.

The operations on iterators are:

Iterator operations	
<code>++p</code>	Pre-increment: make <code>p</code> refer to the next element in the sequence or to one beyond the last element (“advance one element”); the resulting value is <code>p+1</code> .
<code>p++</code>	Post-increment: make <code>p</code> refer to the next element in the sequence or to one beyond the last element (“advance one element”); the resulting value is <code>p</code> (before the increment).
<code>--p</code>	Pre-decrement: make <code>p</code> point to the previous element (“go back one element”); the resulting value is <code>p-1</code> .
<code>p--</code>	Post-decrement: make <code>p</code> point to the previous element (“go back one element”); the resulting value is <code>p</code> (before the decrement).
<code>*p</code>	Access (dereference): <code>*p</code> refers to the element pointed to by <code>p</code> .
<code>p[n]</code>	Access (subscripting): <code>p[n]</code> refers to the element pointed to by <code>p+n</code> ; equivalent to <code>*(p+n)</code> .
<code>p-&gt;m</code>	Access (member access); equivalent to <code>(*p).m</code> .
<code>p==q</code>	Equality: true if <code>p</code> and <code>q</code> point to the same element or both point to one beyond the last element.
<code>p!=q</code>	Inequality: <code>!(p==q)</code> .
<code>p&lt;q</code>	Does <code>p</code> point to an element before what <code>q</code> points to?
<code>p&lt;=q</code>	<code>p&lt;q    p==q</code> .
<code>p&gt;q</code>	Does <code>p</code> point to an element after what <code>q</code> points to?
<code>p&gt;=q</code>	<code>p&gt;q    p==q</code> .
<code>p+=n</code>	Advance <code>n</code> : make <code>p</code> point to the <code>n</code> th element after the one it points to.
<code>p-=n</code>	Advance <code>-n</code> : make <code>p</code> point to the <code>n</code> th element before the one it points to.
<code>q=p+n</code>	<code>q</code> points to the <code>n</code> th element after the one pointed to by <code>p</code> .

### Iterator operations (*continued*)

<b>q=p-n</b>	<b>q</b> points to the <b>n</b> th element before the one pointed to by <b>p</b> ; afterward, we have <b>q+n==p</b> .
<b>advance(p,n)</b>	Like <b>p+=n</b> , but <b>advance()</b> can be used even if <b>p</b> is not a random-access iterator; it may take <b>n</b> steps (through a list).
<b>x=distance(p,q)</b>	Like <b>q-p</b> , but <b>distance()</b> can be used even if <b>p</b> is not a random-access iterator; it may take <b>n</b> steps (through a list).

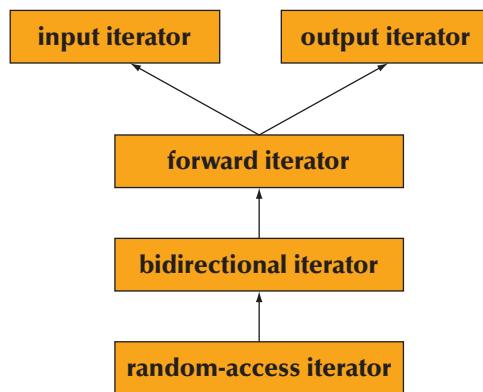
Note that not every kind of iterator (§B.3.2) supports every iterator operation.

### B.3.2 Iterator categories

The standard library provides five kinds of iterators (five “iterator categories”):

Iterator categories	
input iterator	We can iterate forward using <b>++</b> and read each element once only using <b>*</b> . We can compare iterators using <b>==</b> and <b>!=</b> . This is the kind of iterator that <b>istream</b> offers; see §21.7.2.
output iterator	We can iterate forward using <b>++</b> and write each element once only using <b>*</b> . This is the kind of iterator that <b>ostream</b> offers; see §21.7.2.
forward iterator	We can iterate forward repeatedly using <b>++</b> and read and write (unless the elements are <b>const</b> ) elements using <b>*</b> . If it points to a class object, it can use <b>-&gt;</b> to access a member.
bidirectional iterator	We can iterate forward (using <b>++</b> ) and backward (using <b>--</b> ) and read and write (unless the elements are <b>const</b> ) elements using <b>*</b> . This is the kind of iterator that <b>list</b> , <b>map</b> , and <b>set</b> offer.
random-access iterator	We can iterate forward (using <b>++</b> or <b>+=</b> ) and backward (using <b>--</b> or <b>=-</b> ) and read and write (unless the elements are <b>const</b> ) elements using <b>*</b> or <b>[ ]</b> . We can subscript, add an integer to a random-access iterator using <b>+</b> , and subtract an integer using <b>-</b> . We can find the distance between two random-access iterators to the same sequence by subtracting one from the other. We can compare iterators using <b>&lt;</b> , <b>&lt;=</b> , <b>&gt;</b> , and <b>&gt;=</b> . This is the kind of iterator that <b>vector</b> offers.

Logically, these iterators are organized in a hierarchy (§20.8):



Note that since the iterator categories are not classes, this hierarchy is not a class hierarchy implemented using derivation. If you need to do something advanced with iterator categories, look for [iterator\\_traits](#) in an advanced reference.

Each container supplies iterators of a specified category:

- [vector](#) – random access
- [list](#) – bidirectional
- [forward\\_list](#) – forward
- [deque](#) – random access
- [bitset](#) – none
- [set](#) – bidirectional
- [multiset](#) – bidirectional
- [map](#) – bidirectional
- [multimap](#) – bidirectional
- [unordered\\_set](#) – forward
- [unordered\\_multiset](#) – forward
- [unordered\\_map](#) – forward
- [unordered\\_multimap](#) – forward

## B.4 Containers

A container holds a sequence of objects. The elements of the sequence are of the member type called **value\_type**. The most commonly useful containers are:

Sequence containers	
<b>array&lt;T,N&gt;</b>	fixed-size array of <b>N</b> elements of type <b>T</b>
<b>deque&lt;T&gt;</b>	double-ended queue
<b>list&lt;T&gt;</b>	doubly-linked list
<b>forward_list&lt;T&gt;</b>	singly-linked list
<b>vector&lt;T&gt;</b>	dynamic array of elements of type <b>T</b>

Associative containers	
<b>map&lt;K,V&gt;</b>	map from <b>K</b> to <b>V</b> ; a sequence of ( <b>K,V</b> ) pairs
<b>multimap&lt;K,V&gt;</b>	map from <b>K</b> to <b>V</b> ; duplicate keys allowed
<b>set&lt;K&gt;</b>	set of <b>K</b>
<b>multiset&lt;K&gt;</b>	set of <b>K</b> (duplicate keys allowed)
<b>unordered_map&lt;K,V&gt;</b>	map from <b>K</b> to <b>V</b> using a hash function
<b>unordered_multimap&lt;K,V&gt;</b>	map from <b>K</b> to <b>V</b> using a hash function; duplicate keys allowed
<b>unordered_set&lt;K&gt;</b>	set of <b>K</b> using a hash function
<b>unordered_multiset&lt;K&gt;</b>	set of <b>K</b> using a hash function; duplicate keys allowed

The ordered associative containers (**map**, **set**, etc.) have an optional additional template argument specifying the type used for the comparator; for example, **set<K,C>** uses a **C** to compare **K** values.

Container adaptors	
<b>priority_queue&lt;T&gt;</b>	priority queue
<b>queue&lt;T&gt;</b>	queue with <b>push()</b> and <b>pop()</b>
<b>stack&lt;T&gt;</b>	stack with <b>push()</b> and <b>pop()</b>

These containers are defined in `<vector>`, `<list>`, etc. (see §B.1.1). The sequence containers are contiguously allocated or linked lists of elements of their `value_type` (`T` in the notation used above). The associative containers are linked structures (trees) with nodes of their `value_type` (`pair(K,V)` in the notation used above). The sequence of a `set`, `map`, or `multimap` is ordered by its key values (`K`). The sequence of an `unordered_*` does not have a guaranteed order. A `multimap` differs from a `map` in that a key value may occur many times. Container adaptors are containers with specialized operations constructed from other containers.

If in doubt, use `vector`. Unless you have a solid reason not to, use `vector`.

A container uses an “allocator” to allocate and deallocate memory (§19.3.7). We do not cover allocators here; if necessary, see an expert-level reference. By default, an allocator uses `new` and `delete` when it needs to acquire or release memory for its elements.

Where meaningful, an access operation exists in two versions: one for `const` and one for non-`const` objects (§18.5).

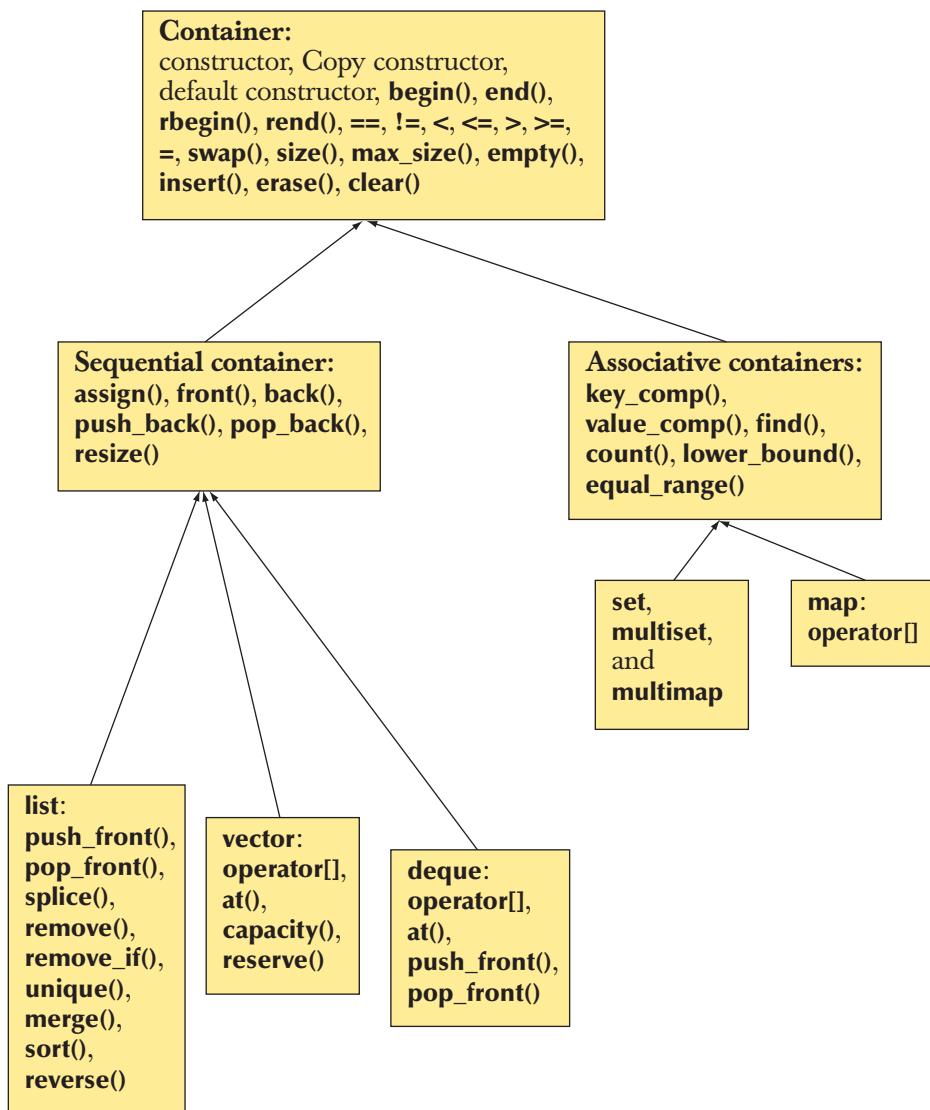
This section lists the common and almost common members of the standard containers. For more details, see Chapter 20. Members that are peculiar to a specific container, such as `list`’s `splice()`, are not listed; see an expert-level reference.

Some data types provide much of what is required from a standard container, but not all. We sometimes refer to those as “almost containers.” The most interesting of those are:

“Almost containers”	
<b><code>T[n]</code></b> built-in array	No <code>size()</code> or other member functions; prefer a container, such as <code>vector</code> , <code>string</code> , or <code>array</code> , over array when you have a choice.
<b><code>string</code></b>	Holds only characters but provides operations useful for text manipulation, such as concatenation ( <code>+</code> and <code>+=</code> ); prefer the standard <code>string</code> to other strings.
<b><code>valarray</code></b>	A numerical vector with vector operations, but with many restrictions to encourage high-performance implementations; use only if you do a lot of vector arithmetic.

### B.4.1 Overview

The operations provided by the standard containers can be summarized like this:



We left out `array` and `forward_list` because they are imperfect fits to the standard library ideal of interchangeability:

- `array` is not a handle, cannot have its number of elements changed after initialization, and must be initialized by an initializer list, rather than by a constructor.
- `forward_list` doesn't support back operations. In particular, it has no `size()`. It is best seen as a container optimized for empty and near-empty sequences.

### B.4.2 Member types

A container defines a set of member types:

Member types	
<code>value_type</code>	type of element
<code>size_type</code>	type of subscripts, element counts, etc.
<code>difference_type</code>	type of difference between iterators
<code>iterator</code>	behaves like <code>value_type*</code>
<code>const_iterator</code>	behaves like <code>const value_type*</code>
<code>reverse_iterator</code>	behaves like <code>value_type*</code>
<code>const_reverse_iterator</code>	behaves like <code>const value_type*</code>
<code>reference</code>	<code>value_type&amp;</code>
<code>const_reference</code>	<code>const value_type&amp;</code>
<code>pointer</code>	behaves like <code>value_type*</code>
<code>const_pointer</code>	behaves like <code>const value_type*</code>
<code>key_type</code>	type of key (associative containers only)
<code>mapped_type</code>	type of mapped value (associative containers only)
<code>key_compare</code>	type of comparison criterion (associative containers only)
<code>allocator_type</code>	type of memory manager

### B.4.3 Constructors, destructors, and assignments

Containers provide a variety of constructors and assignment operations. For a container called **C** (e.g., `vector<double>` or `map<string,int>`) we have:

Constructors, destructors, and assignment	
<b>C c;</b>	<b>c</b> is an empty container.
<b>C{}</b>	Make an empty container.
<b>C c(n);</b>	<b>c</b> initialized with <b>n</b> elements with default element value (not for associative containers).
<b>C c(n,x);</b>	<b>c</b> initialized with <b>n</b> copies of <b>x</b> (not for associative containers).
<b>C c {b,e};</b>	<b>c</b> initialized with elements from <code>[b:e)</code> .
<b>C c {elems};</b>	<b>c</b> initialized with elements from the <code>initializer_list</code> holding <b>elems</b> .
<b>C c {c2};</b>	<b>c</b> is a copy of <b>c2</b> .
<b>~C()</b>	Destroy a <b>C</b> and all of its elements (usually invoked implicitly).
<b>c1=c2</b>	Copy assignment; copy all elements from <b>c2</b> to <b>c1</b> ; after the assignment <b>c1==c2</b> .
<b>c.assign(n,x)</b>	Assign <b>n</b> copies of <b>x</b> (not for associative containers).
<b>c.assign(b,e)</b>	Assign from <code>[b:e)</code> .

Note that for some containers and some element types, a constructor or an element copy may throw an exception.

### B.4.4 Iterators

A container can be viewed as a sequence either in the order defined by the container's **iterator** or in reverse order. For an associative container, the order is based on the container's comparison criterion (by default `<`):

Iterators	
<b>p=c.begin()</b>	<b>p</b> points to the first element of <b>c</b> .
<b>p=c.end()</b>	<b>p</b> points to one past the last element of <b>c</b> .
<b>p=c.rbegin()</b>	<b>p</b> points to the first element of the reverse sequence of <b>c</b> .
<b>p=c.rend()</b>	<b>p</b> points to one past the last element of the reverse sequence of <b>c</b> .

### B.4.5 Element access

Some elements can be accessed directly:

Element access	
<code>c.front()</code>	reference to the first element of <code>c</code>
<code>c.back()</code>	reference to the last element of <code>c</code>
<code>c[i]</code>	reference to element <code>i</code> of <code>c</code> ; unchecked access (not for list)
<code>c.at(i)</code>	reference to element <code>i</code> of <code>c</code> ; checked access ( <code>vector</code> and <code>deque</code> only)

Some implementations – especially debug versions – always do range checking, but you cannot portably rely on that for correctness or on the absence of checking for performance. Where such issues are important, examine your implementations.

### B.4.6 Stack and queue operations

The standard `vector` and `deque` provide efficient operations at the end (back) of their sequence of elements. In addition, `list` and `deque` provide the equivalent operations on the start (front) of their sequences:

Stack and queue operations	
<code>c.push_back(x)</code>	Add <code>x</code> to the end of <code>c</code> .
<code>c.pop_back()</code>	Remove the last element from <code>c</code> .
<code>c.emplace_back(args)</code>	Add <code>T{args}</code> to the end of <code>c</code> ; <code>T</code> is the value type of <code>c</code> .
<code>c.push_front(x)</code>	Add <code>x</code> to <code>c</code> before the first element ( <code>list</code> and <code>deque</code> only).
<code>c.pop_front()</code>	Remove the first element from <code>c</code> ( <code>list</code> and <code>deque</code> only).
<code>c.emplace_front(args)</code>	Add <code>T{args}</code> to <code>c</code> before the first element; <code>T</code> is the value type of <code>c</code> .

Note that `push_front()` and `push_back()` copy an element into a container. This implies that the size of the container increases (by one). If the copy constructor of the element type can throw an exception, a push can fail.

The `push_front()` and `push_back()` operations copy their argument object into the container. For example:

```
vector<pair<string,int>> v;
v.push_back(make_pair("Cambridge",1209));
```

If first creating the object and then copying it seems awkward or potentially inefficient, we can construct the object directly in a newly allocated element slot of the sequence:

```
v.emplace_back("Cambridge",1209);
```

Emplace means “put in place” or “put in position.”

Note that pop operations do not return a value. Had they done so, a copy constructor throwing an exception could have seriously complicated the implementation. Use `front()` and `back()` (§B.4.5) to access stack and queue elements. We have not recorded the complete set of requirements here; feel free to guess (your compiler will usually tell you if you guessed wrong) and to consult more detailed documentation.

### B.4.7 List operations

Containers provide list operations:

List operations	
<code>q=c.insert(p,x)</code>	Add <code>x</code> before <code>p</code> .
<code>q=c.insert(p,n,x)</code>	Add <code>n</code> copies of <code>x</code> before <code>p</code> .
<code>q=c.insert(p,first,last)</code>	Add elements from <code>[first:last)</code> before <code>p</code> .
<code>q=c.emplace(p,args)</code>	Add <code>T{args}</code> before <code>p</code> ; <code>T</code> is the value type of <code>c</code> .
<code>q=c.erase(p)</code>	Remove element at <code>p</code> from <code>c</code> .
<code>q=c.erase(first,last)</code>	Erase <code>[first:last)</code> of <code>c</code> .
<code>c.clear()</code>	Erase all elements of <code>c</code> .

For `insert()` functions, the result, `q`, points to the last element inserted. For `erase()` functions, `q` points to the element that followed the last element erased.

### B.4.8 Size and capacity

The size is the number of elements in the container; the capacity is the number of elements that a container can hold before allocating more memory:

Size and capacity	
<code>x=c.size()</code>	<code>x</code> is the number of elements of <code>c</code> .
<code>c.empty()</code>	Is <code>c</code> empty?

### Size and capacity (*continued*)

<code>x=c.max_size()</code>	<code>x</code> is the largest possible number of elements of <code>c</code> .
<code>x=c.capacity()</code>	<code>x</code> is the space allocated for <code>c</code> ( <code>vector</code> and <code>string</code> only).
<code>c.reserve(n)</code>	Reserve space for <code>n</code> elements for <code>c</code> ( <code>vector</code> and <code>string</code> only).
<code>c.resize(n)</code>	Change the size of <code>c</code> to <code>n</code> ( <code>vector</code> , <code>string</code> , <code>list</code> , and <code>deque</code> only).

When changing the size or the capacity, the elements may be moved to new storage locations. That implies that iterators (and pointers and references) to elements may become invalid (e.g., point to the old element locations).

### B.4.9 Other operations

Containers can be copied (see §B.4.3), compared, and swapped:

### Comparisons and swap

<code>c1==c2</code>	Do all corresponding elements of <code>c1</code> and <code>c2</code> compare equal?
<code>c1!=c2</code>	Do any corresponding elements of <code>c1</code> and <code>c2</code> compare not equal?
<code>c1&lt;c2</code>	Is <code>c1</code> lexicographically before <code>c2</code> ?
<code>c1&lt;=c2</code>	Is <code>c1</code> lexicographically before or equal to <code>c2</code> ?
<code>c1&gt;c2</code>	Is <code>c1</code> lexicographically after <code>c2</code> ?
<code>c1&gt;=c2</code>	Is <code>c1</code> lexicographically after or equal to <code>c2</code> ?
<code>swap(c1,c2)</code>	Swap elements of <code>c1</code> and <code>c2</code> .
<code>c1.swap(c2)</code>	Swap elements of <code>c1</code> and <code>c2</code> .

When comparing containers with an operator (e.g., `<`), their elements are compared using the equivalent element operator (i.e., `<`).

### B.4.10 Associative container operations

Associative containers provide lookup based on keys:

### Associative container operations

<code>c[k]</code>	Refers to the element with key <code>k</code> (containers with unique keys).
<code>p=c.find(k)</code>	<code>p</code> points to the first element with key <code>k</code> .
<code>p=c.lower_bound(k)</code>	<code>p</code> points to the first element with key <code>k</code> .

### Associative container operations (*continued*)

<code>p=c.upper_bound(k)</code>	<code>p</code> points to the first element with key greater than <code>k</code> .
<code>pair(p1,p2)=c.equal_range(k)</code>	<code>[p1:p2)</code> are the elements with key <code>k</code> .
<code>r=c.key_comp()</code>	<code>r</code> is a copy of the key-comparison object.
<code>r=c.value_comp()</code>	<code>r</code> is a copy of the <code>mapped_value</code> -comparison object. If a key is not found, <code>c.end()</code> is returned.

The first iterator of the pair returned by `equal_range` is `lower_bound` and the second is `upper_bound`. You can print the value of all elements with the key "Marian" in a `multimap<string,int>` like this:

```
string k = "Marian";
auto pp = m.equal_range(k);
if (pp.first!=pp.second)
    cout << "elements with value '" << k << "' :\n";
else
    cout << "no element with value '" << k << "'\n";
for (auto p = pp.first; p!=pp.second; ++p)
    cout << p->second << '\n';
```

We could equivalently have used

```
auto pp = make_pair(m.lower_bound(k),m.upper_bound(k));
```

However, that would take about twice as long to execute. The `equal_range`, `lower_bound`, and `upper_bound` algorithms are also provided for sorted sequences (§B.5.4). The definition of `pair` is in §B.6.3.

## B.5 Algorithms

There are about 60 standard algorithms defined in `<algorithm>`. They all operate on sequences defined by a pair of iterators (for inputs) or a single iterator (for outputs).

When copying, comparing, etc. two sequences, the first is represented by a pair of iterators `[b:e)` but the second by just a single iterator, `b2`, which is considered the start of a sequence holding sufficient elements for the algorithm, for example, as many elements as the first sequence: `[b2:b2+(e-b))`.

Some algorithms, such as `sort`, require random-access iterators, whereas many, such as `find`, only read their elements in order so that they can make do with a forward iterator.

Many algorithms follow the usual convention of returning the end of a sequence to represent “not found.” We don’t mention that for each algorithm.

### B.5.1 Nonmodifying sequence algorithms

A nonmodifying algorithm just reads the elements of a sequence; it does not rearrange the sequence and does not change the value of the elements:

Nonmodifying sequence algorithms	
<code>f=for_each(b,e,f)</code>	Do <code>f</code> for each element in <code>[b:e)</code> ; return <code>f</code> .
<code>p=find(b,e,v)</code>	<code>p</code> points to the first occurrence of <code>v</code> in <code>[b:e)</code> .
<code>p=find_if(b,e,f)</code>	<code>p</code> points to the first element in <code>[b:e)</code> so that <code>f(*p)</code> .
<code>p=find_first_of(b,e,b2,e2)</code>	<code>p</code> points to the first element in <code>[b:e)</code> so that <code>*p==*q</code> for some <code>q</code> in <code>[b2:e2)</code> .
<code>p=find_first_of(b,e,b2,e2,f)</code>	<code>p</code> points to the first element in <code>[b:e)</code> so that <code>f(*p,*q)</code> for some <code>q</code> in <code>[b2:e2)</code> .
<code>p=adjacent_find(b,e)</code>	<code>p</code> points to the first <code>p</code> in <code>[b:e)</code> such that <code>*p==*(p+1)</code> .
<code>p=adjacent_find(b,e,f)</code>	<code>p</code> points to the first <code>p</code> in <code>[b:e)</code> such that <code>f(*p,*(p+1))</code> .
<code>equal(b,e,b2)</code>	Do all elements of <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> compare equal?
<code>equal(b,e,b2,f)</code>	Do all elements of <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> compare equal using <code>f(*p,*q)</code> as the test?
<code>pair(p1,p2)=mismatch(b,e,b2)</code>	<code>(p1,p2)</code> points to the first pair of elements in <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> for which <code>!(*p1==*p2)</code> .
<code>pair(p1,p2)=mismatch(b,e,b2,f)</code>	<code>(p1,p2)</code> points to the first pair of elements in <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> for which <code>!f(*p1,*p2)</code> .
<code>p=search(b,e,b2,e2)</code>	<code>p</code> points to the first <code>*p</code> in <code>[b:e)</code> such that <code>*p</code> equals an element in <code>[b2:e2)</code> .
<code>p=search(b,e,b2,e2,f)</code>	<code>p</code> points to the first <code>*p</code> in <code>[b:e)</code> such that <code>f(*p,*q)</code> for an element <code>*q</code> in <code>[b2:e2)</code> .
<code>p=find_end(b,e,b2,e2)</code>	<code>p</code> points to the last <code>*p</code> in <code>[b:e)</code> such that <code>*p</code> equals an element in <code>[b2:e2)</code> .
<code>p=find_end( b,e,b2,e2,f)</code>	<code>p</code> points to the last <code>*p</code> in <code>[b:e)</code> such that <code>f(*p,*q)</code> for an element <code>*q</code> in <code>[b2:e2)</code> .

### Nonmodifying sequence algorithms (*continued*)

<code>p=search_n(b,e,n,v)</code>	<code>p</code> points to the first element of <code>[b:e]</code> such that each element in <code>[p:p+n]</code> has the value <code>v</code> .
<code>p=search_n(b,e,n,v,f)</code>	<code>p</code> points to the first element of <code>[b:e]</code> such that for each element <code>*q</code> in <code>[p:p+n]</code> we have <code>f(*q,v)</code> .
<code>x=count(b,e,v)</code>	<code>x</code> is the number of occurrences of <code>v</code> in <code>[b:e]</code> .
<code>x=count_if(b,e,v,f)</code>	<code>x</code> is the number of elements in <code>[b:e]</code> so that <code>f(*p,v)</code> .

Note that nothing stops the operation passed to `for_each` from modifying elements; that's considered acceptable. Passing an operation that changes the elements it examines to some other algorithm (e.g., `count` or `==`) is not acceptable.

An example (of proper use):

```
bool odd(int x) { return x&1; }

int n_even(const vector<int>& v)      // count the number of even values in v
{
    return v.size() - count_if(v.begin(), v.end(), odd);
}
```

### B.5.2 Modifying sequence algorithms

The modifying algorithms (also called *mutating sequence algorithms*) can (and often do) modify the elements of their argument sequences.

#### Modifying sequence algorithms

<code>p=transform(b,e,out,f)</code>	Apply <code>*p2=f(*p1)</code> to every <code>*p1</code> in <code>[b:e]</code> , writing to the corresponding <code>*p2</code> in <code>[out:out+(e-b)]</code> ; <code>p=out+(e-b)</code> .
<code>p=transform(b,e,b2,out,f)</code>	Apply <code>*p3=f(*p1,*p2)</code> to every element in <code>*p1</code> in <code>[b:e]</code> and the corresponding element <code>*p2</code> in <code>[b2:b2+(e-b)]</code> , writing to <code>*p3</code> in <code>[out:out+(e-b)]</code> ; <code>p=out+(e-b)</code> .
<code>p=copy(b,e,out)</code>	Copy <code>[b:e]</code> to <code>[out:p]</code> .
<code>p=copy_backward(b,e,out)</code>	Copy <code>[b:e]</code> to <code>[out:p]</code> starting with its last element.

Modifying sequence algorithms ( <i>continued</i> )	
<code>p=unique(b,e)</code>	Move elements in $[b:e]$ so that $[b:p]$ has adjacent duplicates removed ( <code>==</code> defines “duplicate”).
<code>p=unique(b,e,f)</code>	Move elements in $[b:e]$ so that $[b:p]$ has adjacent duplicates removed ( <code>f</code> defines “duplicate”).
<code>p=unique_copy(b,e,out)</code>	Copy $[b:e]$ to $[out:p]$ ; don’t copy adjacent duplicates.
<code>p=unique_copy(b,e,out,f)</code>	Copy $[b:e]$ to $[out:p]$ ; don’t copy adjacent duplicates ( <code>f</code> defines “duplicate”).
<code>replace(b,e,v,v2)</code>	Replace elements $*q$ in $[b:e]$ for which $*q==v$ with $v2$ .
<code>replace(b,e,f,v2)</code>	Replace elements $*q$ in $[b:e]$ for which $f(*q)$ with $v2$ .
<code>p=replace_copy(b,e,out,v,v2)</code>	Copy $[b:e]$ to $[out:p]$ , replacing elements $*q$ in $[b:e]$ for which $*q==v$ with $v2$ .
<code>p=replace_copy(b,e,out,f,v2)</code>	Copy $[b:e]$ to $[out:p]$ , replacing elements $*q$ in $[b:e]$ for which $f(*q)$ with $v2$ .
<code>p=remove(b,e,v)</code>	Move elements $*q$ in $[b:e]$ so that $[b:p]$ becomes the elements for which $!(*q==v)$ .
<code>p=remove(b,e,v,f)</code>	Move elements $*q$ in $[b:e]$ so that $[b:p]$ becomes the elements for which $!f(*q)$ .
<code>p=remove_copy(b,e,out,v)</code>	Copy elements from $[b:e]$ for which $!(*q==v)$ to $[out:p]$ .
<code>p=remove_copy_if(b,e,out,f)</code>	Copy elements from $[b:e]$ for which $!f(*q,v)$ to $[out:p]$ .
<code>reverse(b,e)</code>	Reverse the order of elements in $[b:e]$ .
<code>p=reverse_copy(b,e,out)</code>	Copy $[b:e]$ into $[out:p]$ in reverse order.
<code>rotate(b,m,e)</code>	Rotate elements: treat $[b:e]$ as a circle with the first element right after the last. Move $*b$ to $*m$ and in general move $(b+i)$ to $((b+(i+(e-m))\%(e-b))$ .
<code>p=rotate_copy(b,m,e,out)</code>	Copy $[b:e]$ into a rotated sequence $[out:p]$ .
<code>random_shuffle(b,e)</code>	Shuffle elements of $[b:e]$ into a distribution using the default uniform random number generator.
<code>random_shuffle(b,e,f)</code>	Shuffle elements of $[b:e]$ into a distribution using <code>f</code> as a random number generator.

A shuffle algorithm shuffles its sequence much in the way we would shuffle a pack of cards; that is, after a shuffle, the elements are in a random order, where “random” is defined by the distribution produced by the random number generator.

Please note that these algorithms do not know if their argument sequence is a container, so they do not have the ability to add or remove elements. Thus, an algorithm such as `remove` cannot shorten its input sequence by deleting (erasing) elements; instead, it (re)moves the elements it keeps to the front of the sequence:

```
template<typename Iter>
void print_digits(const string& s, Iter b, Iter e)
{
    cout << s;
    while (b!=e) { cout << *b; ++b; }
    cout << '\n';
}

void ff()
{
    vector<int> v {1,1,1, 2,2, 3, 4,4,4, 3,3,3, 5,5,5,5, 1,1,1};
    print_digits("all: ",v.begin(), v.end());

    auto pp = unique(v.begin(),v.end());
    print_digits("head: ",v.begin(),pp);
    print_digits("tail: ",pp,v.end());

    pp=remove(v.begin(),pp,4);
    print_digits("head: ",v.begin(),pp);
    print_digits("tail: ",pp,v.end());
}
```

The resulting output is

```
all: 1112234443335555111
head: 1234351
tail: 443335555111
head: 123351
tail: 1443335555111
```

### B.5.3 Utility algorithms

Technically, these utility algorithms are also modifying sequence algorithms, but we thought it a good idea to list them separately, lest they get overlooked.

Utility algorithms	
<code>swap(x,y)</code>	Swap <code>x</code> and <code>y</code> .
<code>iter_swap(p,q)</code>	Swap <code>*p</code> and <code>*q</code> .
<code>swap_ranges(b,e,b2)</code>	Swap the elements of <code>[b:e)</code> and <code>[b2:b2+(e-b))</code> .
<code>fill(b,e,v)</code>	Assign <code>v</code> to every element of <code>[b:e)</code> .
<code>fill_n(b,n,v)</code>	Assign <code>v</code> to every element of <code>[b:b+n)</code> .
<code>generate(b,e,f)</code>	Assign <code>f()</code> to every element of <code>[b:e)</code> .
<code>generate_n(b,n,f)</code>	Assign <code>f()</code> to every element of <code>[b:b+n)</code> .
<code>uninitialized_fill(b,e,v)</code>	Initialize all elements in <code>[b:e)</code> with <code>v</code> .
<code>uninitialized_copy(b,e,out)</code>	Initialize all elements of <code>[out:out+(e-b))</code> with the corresponding element from <code>[b:e)</code> .

Note that uninitialized sequences should occur only at the lowest level of programming, usually inside the implementation of containers. Elements that are targets of `uninitialized_fill` or `uninitialized_copy` must be of built-in type or uninitialized.

#### B.5.4 Sorting and searching

Sorting and searching are fundamental and the needs of programmers are quite varied. Comparison is by default done using the `<` operator, and equivalence of a pair of values `a` and `b` is determined by `!(a < b) && !(b < a)` rather than requiring operator `==`.

Sorting and searching	
<code>sort(b,e)</code>	Sort <code>[b:e)</code> .
<code>sort(b,e,f)</code>	Sort <code>[b:e)</code> using <code>f(*p,*q)</code> as the sorting criterion.
<code>stable_sort(b,e)</code>	Sort <code>[b:e)</code> , maintaining the order of equivalent elements.
<code>stable_sort(b,e,f)</code>	Sort <code>[b:e)</code> using <code>f(*p,*q)</code> as the sorting criterion, maintaining the order of equivalent elements.
<code>partial_sort(b,m,e)</code>	Sort <code>[b:e)</code> to get <code>[b:m)</code> into order; <code>[m:e)</code> need not be sorted.
<code>partial_sort(b,m,e,f)</code>	Sort <code>[b:e)</code> using <code>f(*p,*q)</code> as the sorting criterion to get <code>[b:m)</code> into order; <code>[m:e)</code> need not be sorted.

Sorting and searching (continued)	
<b>partial_sort_copy(b,e,b2,e2)</b>	Sort enough of <code>[b:e)</code> to copy the <code>e2–b2</code> first elements to <code>[b2:e2)</code> .
<b>partial_sort_copy(b,e,b2,e2,f)</b>	Sort enough of <code>[b:e)</code> to copy the <code>e2–b2</code> first elements to <code>[b2:e2)</code> ; use <code>f</code> as the comparison.
<b>nth_element(b,e)</b>	Put the <code>nth</code> element of <code>[b:e)</code> in its proper place.
<b>nth_element(b,e,f)</b>	Put the <code>nth</code> element of <code>[b:e)</code> in its proper place using <code>f</code> for comparison.
<b>p=lower_bound(b,e,v)</b>	<code>p</code> points to the first occurrence of <code>v</code> in <code>[b:e)</code> .
<b>p=lower_bound(b,e,v,f)</b>	<code>p</code> points to the first occurrence of <code>v</code> in <code>[b:e)</code> using <code>f</code> for comparison.
<b>p=upper_bound(b,e,v)</b>	<code>p</code> points to the first value larger than <code>v</code> in <code>[b:e)</code> .
<b>p=upper_bound(b,e,v,f)</b>	<code>p</code> points to the first value larger than <code>v</code> in <code>[b:e)</code> using <code>f</code> for comparison.
<b>binary_search(b,e,v)</b>	Is <code>v</code> in the sorted sequence <code>[b:e)</code> ?
<b>binary_search(b,e,v,f)</b>	Is <code>v</code> in the sorted sequence <code>[b:e)</code> using <code>f</code> for comparison?
<b>pair(p1,p2)=equal_range(b,e,v)</b>	<code>[p1,p2)</code> is the subsequence of <code>[b:e)</code> with the value <code>v</code> ; basically, a binary search for <code>v</code> .
<b>pair(p1,p2)=equal_range(b,e,v,f)</b>	<code>[p1,p2)</code> is the subsequence of <code>[b:e)</code> with the value <code>v</code> using <code>f</code> for comparison; basically, a binary search for <code>v</code> .
<b>p=merge(b,e,b2,e2,out)</b>	Merge two sorted sequences <code>[b2:e2)</code> and <code>[b:e)</code> into <code>[out:p)</code> .
<b>p=merge(b,e,b2,e2,out,f)</b>	Merge two sorted sequences <code>[b2:e2)</code> and <code>[b:e)</code> into <code>[out,out+p)</code> using <code>f</code> as the comparison.
<b>inplace_merge(b,m,e)</b>	Merge two sorted subsequences <code>[b:m)</code> and <code>[m:e)</code> into a sorted sequence <code>[b:e)</code> .
<b>inplace_merge(b,m,e,f)</b>	Merge two sorted subsequences <code>[b:m)</code> and <code>[m:e)</code> into a sorted sequence <code>[b:e)</code> using <code>f</code> as the comparison.
<b>p=partition(b,e,f)</b>	Place elements for which <code>f(*p1)</code> in <code>[b:p)</code> and other elements in <code>[p:e)</code> .
<b>p=stable_partition(b,e,f)</b>	Place elements for which <code>f(*p1)</code> in <code>[b:p)</code> and other elements in <code>[p:e)</code> , preserving relative order.

For example:

```
vector<int> v {3,1,4,2};
list<double> lst {0.5,1.5,3,2.5};      // lst is in order
sort(v.begin(),v.end());                  // put v in order
vector<double> v2;
merge(v.begin(),v.end(),lst.begin(),lst.end(),back_inserter(v2));
for (auto x : v2) cout << x << ", ";
```

For inserters, see §B.6.1. The output is

**0.5, 1, 1.5, 2, 2, 2.5, 3, 4,**

The **equal\_range**, **lower\_bound**, and **upper\_bound** algorithms are used just like their equivalents for associative containers; see §B.4.10.

### B.5.5 Set algorithms

These algorithms treat a sequence as a set of elements and provide the basic set operations. The input sequences are supposed to be sorted and the output sequences are also sorted:

<b>Set algorithms</b>	
<b>includes(b,e,b2,e2)</b>	Are all elements of [b2:e2) also in [b:e)?
<b>includes(b,e,b2,e2,f)</b>	Are all elements of [b2:e2) also in [b:e) using f for comparison?
<b>p=set_union(b,e,b2,e2,out)</b>	Construct a sorted sequence [out:p) of elements that are in either [b:e) or [b2:e2).
<b>p=set_union(b,e,b2,e2,out,f)</b>	Construct a sorted sequence [out:p) of elements that are in either [b:e) or [b2:e2) using f for comparison.
<b>p=set_intersection(b,e,b2,e2,out)</b>	Construct a sorted sequence [out:p) of elements that are in both [b:e) and [b2:e2).
<b>p=set_intersection(b,e,b2,e2,out,f)</b>	Construct a sorted sequence [out:p) of elements that are in both [b:e) and [b2:e2) using f for comparison.
<b>p=set_difference(b,e,b2,e2,out)</b>	Construct a sorted sequence [out:p) of elements that are in [b:e) but not in [b2:e2).

Set algorithms ( <i>continued</i> )	
<code>p=set_difference(b,e,b2,e2,out,f)</code>	Construct a sorted sequence [ <code>out:p</code> ) of elements that are in [ <code>b:e</code> ) but not in [ <code>b2:e2</code> ) using <code>f</code> for comparison.
<code>p=set_symmetric_difference(b,e,b2,e2,out)</code>	Construct a sorted sequence [ <code>out:p</code> ) of elements that are in [ <code>b:e</code> ) or [ <code>b2:e2</code> ) but not in both.
<code>p=set_symmetric_difference(b,e,b2,e2,out,f)</code>	Construct a sorted sequence [ <code>out:p</code> ) of elements that are in [ <code>b:e</code> ) or [ <code>b2:e2</code> ) but not in both using <code>f</code> for comparison.

### B.5.6 Heaps

A heap is a data structure that keeps the element with highest value first. The heap algorithms allow a programmer to treat a random-access sequence as a heap:

Heap operations	
<code>make_heap(b,e)</code>	Make the sequence ready to be used as a heap.
<code>make_heap(b,e,f)</code>	Make the sequence ready to be used as a heap, using <code>f</code> for comparison.
<code>push_heap(b,e)</code>	Add an element to the heap (in its proper place).
<code>push_heap(b,e,f)</code>	Add an element to the heap, using <code>f</code> for comparison.
<code>pop_heap(b,e)</code>	Remove the largest (first) element from the heap.
<code>pop_heap(b,e,f)</code>	Remove an element from the heap, using <code>f</code> for comparison.
<code>sort_heap(b,e)</code>	Sort the heap.
<code>sort_heap(b,e,f)</code>	Sort the heap, using <code>f</code> for comparison.

The point of a heap is to provide fast addition of elements and fast access to the element with the highest value. The main use of heaps is to implement priority queues.

### B.5.7 Permutations

Permutations are used to generate combinations of elements of a sequence. For example, the permutations of `abc` are `abc`, `acb`, `bac`, `bca`, `cab`, and `cba`.

Permutations	
<code>x=next_permutation(b,e)</code>	Make <code>[b:e)</code> the next permutation in lexicographical order.
<code>x=next_permutation(b,e,f)</code>	Make <code>[b:e)</code> the next permutation in lexicographical order, using <code>f</code> for comparison.
<code>x=prev_permutation(b,e)</code>	Make <code>[b:e)</code> the previous permutation in lexicographical order.
<code>x=prev_permutation(b,e,f)</code>	Make <code>[b:e)</code> the previous permutation in lexicographical order, using <code>f</code> for comparison.

The return value (`x`) for `next_permutation` is `false` if `[b:e)` already contains the last permutation (`cba` in the example); in that case, it returns the first permutation (`abc` in the example). The return value for `prev_permutation` is `false` if `[b:e)` already contains the first permutation (`abc` in the example); in that case, it returns the last permutation (`cba` in the example).

### B.5.8 min and max

Value comparisons are useful in many contexts:

min and max	
<code>x=max(a,b)</code>	<code>x</code> is the larger of <code>a</code> and <code>b</code> .
<code>x=max(a,b,f)</code>	<code>x</code> is the larger of <code>a</code> and <code>b</code> using <code>f</code> for comparison.
<code>x=max({elems})</code>	<code>x</code> is the largest element in <code>{elems}</code> .
<code>x=max({elems},f)</code>	<code>x</code> is the largest element in <code>{elems}</code> using <code>f</code> for comparison.
<code>x=min(a,b)</code>	<code>x</code> is the smaller of <code>a</code> and <code>b</code> .
<code>x=min(a,b,f)</code>	<code>x</code> is the smaller of <code>a</code> and <code>b</code> using <code>f</code> for comparison.
<code>x=min({elems})</code>	<code>x</code> is the smallest element in <code>{elems}</code> .
<code>x=min({elems},f)</code>	<code>x</code> is the smallest element in <code>{elems}</code> using <code>f</code> for comparison.
<code>pair(x,y)=minmax(a,b)</code>	<code>x</code> is <code>max(a,b)</code> and <code>y</code> is <code>min(a,b)</code> .
<code>pair(x,y)=minmax(a,b,f)</code>	<code>x</code> is <code>max(a,b,f)</code> and <code>y</code> is <code>min(a,b,f)</code> .
<code>pair(x,y)=minmax({elems})</code>	<code>x</code> is <code>max({elems})</code> and <code>y</code> is <code>min({elems})</code> .
<code>pair(x,y)=minmax({elems},f)</code>	<code>x</code> is <code>max({elems},f)</code> and <code>y</code> is <code>min({elems},f)</code> .

<b>min and max (continued)</b>	
<b>p=max_element(b,e)</b>	<b>p</b> points to the largest element of [b:e).
<b>p=max_element(b,e,f)</b>	<b>p</b> points to the largest element of [b:e] using <b>f</b> for the element comparison.
<b>p=min_element(b,e)</b>	<b>p</b> points to the smallest element of [b:e).
<b>p=min_element(b,e,f)</b>	<b>p</b> points to the smallest element of [b:e) using <b>f</b> for the element comparison.
<b>lexicographical_compare(b,e,b2,e2)</b>	Is [b:e]<[b2:e2]?
<b>lexicographical_compare(b,e,b2,e2,f)</b>	Is [b:e]<[b2:e2], using <b>f</b> for the element comparison?

## B.6 STL utilities

The standard library provides a few facilities for making it easier to use standard library algorithms.

### B.6.1 Inserters

Producing output through an iterator into a container implies that elements pointed to by the iterator and following it can be overwritten. This also implies the possibility of overflow and consequent memory corruption. For example:

```
void f(vector<int>& vi)
{
    fill_n(vi.begin(), 200, 7);      // assign 7 to vi[0]..[199]
}
```

If **vi** has fewer than 200 elements, we are in trouble.

In **<iterator>**, the standard library provides three iterators to deal with this problem by adding (inserting) elements to a container rather than overwriting old elements. Three functions are provided for generating those inserting iterators:

<b>Inserters</b>	
<b>r=back_inserter(c)</b>	* <b>r=x</b> causes a <b>c.push_back(x)</b> .
<b>r=front_inserter(c)</b>	* <b>r=x</b> causes a <b>c.push_front(x)</b> .
<b>r= inserter(c,p)</b>	* <b>r=x</b> causes a <b>c.insert(p,x)</b> .

For **inserter(c,p)**, **p** must be a valid iterator for the container **c**. Naturally, a container grows by one element each time a value is written to it through an insert iterator. When written to, an inserter inserts a new element into a sequence using **push\_back(x)**, **c.push\_front()**, or **insert()** rather than overwriting an existing element. For example:

```
void g(vector<int>& vi)
{
    fill_n(back_inserter(vi), 200,7);      // add 200 7s to the end of vi
}
```

## B.6.2 Function objects

Many of the standard algorithms take function objects (or functions) as arguments to control the way they work. Common uses are comparison criteria, predicates (functions returning **bool**), and arithmetic operations. In **<functional>**, the standard library supplies a few common function objects.

Predicates	
<b>p=equal_to&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x==y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=not_equal_to&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x!=y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=greater&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x&gt;y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=less&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x&lt;y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=greater_equal&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x&gt;=y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=less_equal&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x&lt;=y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=logical_and&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x&amp;&amp;y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=logical_or&lt;T&gt;{}</b>	<b>p(x,y)</b> means <b>x  y</b> when <b>x</b> and <b>y</b> are of type <b>T</b> .
<b>p=logical_not&lt;T&gt;{}</b>	<b>p(x)</b> means <b>!x</b> when <b>x</b> is of type <b>T</b> .

For example:

```
vector<int> v;
// . . .
sort(v.begin(),v.end(),greater<int>());      // sort v in decreasing order
```

Note that **logical\_and** and **logical\_or** always evaluate both their arguments (whereas **&&** and **||** do not).

Also, a lambda expression (§15.3.3) is often an alternative to a simple function object:

```
sort(v.begin(),v.end(),[](int x, int y) { return x>y;}); // sort v in decreasing order
```

Arithmetic operations	
<code>f=plus&lt;T&gt;{}</code>	<code>f(x,y)</code> means $x+y$ when <code>x</code> and <code>y</code> are of type <code>T</code> .
<code>f=minus&lt;T&gt;{}</code>	<code>f(x,y)</code> means $x-y$ when <code>x</code> and <code>y</code> are of type <code>T</code> .
<code>f=multiples&lt;T&gt;{}</code>	<code>f(x,y)</code> means $x*y$ when <code>x</code> and <code>y</code> are of type <code>T</code> .
<code>f=divides&lt;T&gt;{}</code>	<code>f(x,y)</code> means $x/y$ when <code>x</code> and <code>y</code> are of type <code>T</code> .
<code>f=modulus&lt;T&gt;{}</code>	<code>f(x,y)</code> means $x \% y$ when <code>x</code> and <code>y</code> are of type <code>T</code> .
<code>f=negate&lt;T&gt;{}</code>	<code>f(x)</code> means $-x$ when <code>x</code> is of type <code>T</code> .

Adaptors	
<code>f=bind(g,args)</code>	<code>f(x)</code> means <code>g(x,args)</code> where <code>args</code> can be one or more arguments.
<code>f=mem_fn(mf)</code>	<code>f(p,args)</code> means <code>p-&gt;mf(args)</code> where <code>args</code> can be one or more arguments.
<code>Function&lt;F&gt; f {g}</code>	<code>f(args)</code> means <code>g(args)</code> where <code>args</code> can be one or more arguments. <code>F</code> is the type of <code>g</code> .
<code>f=not1(g)</code>	<code>f(x)</code> means <code>!g(x)</code> .
<code>f=not2(g)</code>	<code>f(x,y)</code> means <code>!g(x,y)</code> .

Note that `function` is a template, so that you can define variables of type `function<T>` and assign callable objects to such variables. For example:

```
int f1(double);
function<int(double)> fct {f1};      // initialize to f1
int x = fct(2.3);                      // call f1(2.3)
function<int(double)> fun;             // fun can hold any int(double)
fun = f1;
```

### B.6.3 pair and tuple

In `<utility>`, the standard library provides a few “utility components,” including `pair`:

```
template <class T1, class T2>
struct pair {
    typedef T1 first_type;
    typedef T2 second_type;
    T1 first;
    T2 second;

    // ... copy and move operations ...
};

template <class T1, class T2>
constexpr pair<T1,T2> make_pair(T1 x, T2 y) { return pair<T1,T2>(x,y); }
```

The `make_pair()` function makes the use of pairs simple. For example, here is the outline of a function that returns a value and an error indicator:

```
pair<double,error_indicator> my_fct(double d)
{
    errno = 0;      // clear C-style global error indicator
    // ... do a lot of computation involving d computing x ...
    error_indicator ee = errno;
    errno = 0;      // clear C-style global error indicator
    return make_pair(x,ee);
}
```

This example of a useful idiom can be used like this:

```
pair<int,error_indicator> res = my_fct(123.456);
if (res.second==0) {
    // ... use res.first ...
}
else {
    // oops: error
}
```

We use **pair** when we need exactly two elements and don't care to define a specific type. If we need zero or more elements, we can use a **tuple** from **<tuple>**:

```
template <typename... Types>
struct tuple {
    explicit constexpr tuple(const Types& ...); // construct from N values
    template<typename... Atypes>
    explicit constexpr tuple(const Atypes&& ...); // construct from N values

    // ... copy and move operations ...
};

template <class... Types>
constexpr tuple<Types...> make_tuple(Types&&...); // construct tuple
// from N values
```

The **tuple** implementation uses a feature beyond the scope of this book, variadic templates. This is what those ellipses (...) refer to. However, we can use **tuples** much as we do **pairs**. For example:

```
auto t0 = make_tuple(); // no elements
auto t1 = make_tuple(123.456); // one element of type double
auto t2 = make_tuple(123.456, 'a'); // two elements of types double and char
auto t3 = make_tuple(12,'a',string{"How?"}); // three elements of types int,
// char, and string
```

A **tuple** can have many elements, so we can't just use **first** and **second** to access them. Instead, a function **get** is used:

```
auto d = get<0>(t1); // the double
auto n = get<0>(t3); // the int
auto c = get<1>(t3); // the char
auto s = get<2>(t3); // the string
```

The subscript for **get** is provided as a template argument. As can be seen from the example, **tuple** subscripting is zero-based.

Tuples are mostly used in generic programming.

#### B.6.4 initializer\_list

In **<initializer\_list>**, we find the definition of **initializer\_list**:

```
template<typename T>
class initializer_list {
```

```

public:
    initializer_list() noexcept;

    size_t size() const noexcept;           // number of elements
    const T* begin() const noexcept;        // first element
    const T* end() const noexcept;          // one-past-last element

    // ...
};

```

When the compiler sees a `{}` initializer list with elements of type `X`, that list is used to construct an `initializer_list<X>` (§14.2.1, §18.2). Unfortunately, `initializer_list` does not support the subscript operator (`[ ]`).

### B.6.5 Resource management pointers

A built-in pointer does not indicate whether it represents ownership of the object it points to. That can seriously complicate programming (§19.5). The resource management pointers `unique_ptr` and `shared_ptr` are defined in `<memory>` to deal with that problem:

- `unique_ptr` (§19.5.4) represents exclusive ownership; there can be only one `unique_ptr` to an object and the object is deleted when its `unique_ptr` is destroyed.
- `shared_ptr` represents shared ownership; there can be many `shared_ptr`s to an object, and the object is deleted when its last `shared_ptr` is destroyed.

<code>unique_ptr&lt;p&gt;</code> (simplified)	
<code>unique_ptr up {};</code>	Default constructor: <code>up</code> holds the <code>nullptr</code> .
<code>unique_ptr up {p};</code>	<code>up</code> holds <code>p</code> .
<code>unique_ptr up {up2};</code>	Move constructor: <code>up</code> holds <code>up2</code> 's <code>p</code> ; <code>up2</code> holds the <code>nullptr</code> .
<code>up.reset()</code>	Delete the pointer <code>up</code> holds.
<code>p=up.get()</code>	<code>p</code> is the pointer held by <code>up</code> .
<code>p=up.release()</code>	<code>p</code> is the pointer held by <code>up</code> ; <code>up</code> holds the <code>nullptr</code> .
<code>up.reset(p)</code>	Delete the pointer held by <code>up</code> ; <code>up</code> holds <code>p</code> .
<code>up=make_unique&lt;X&gt;(args)</code>	<code>up</code> holds <code>new&lt;X&gt;(args)</code> (C++14).

The usual pointer operations, such as `*`, `->`, `==`, and `<`, can be used for `unique_ptr`s. Additionally, a `unique_ptr` can be defined to use a delete action different from plain `delete`.

<code>shared_ptr&lt;p&gt;</code> (simplified)	
<code>shared_ptr sp {};</code>	Default constructor: <code>sp</code> holds the <code>nullptr</code> .
<code>shared_ptr sp {p};</code>	<code>sp</code> holds <code>p</code> .
<code>shared_ptr sp {sp2};</code>	Copy constructor: <code>sp</code> and <code>sp2</code> both hold <code>sp2</code> 's <code>p</code> .
<code>shared_ptr sp {move(sp2)};</code>	Move constructor: <code>sp</code> holds <code>sp2</code> 's <code>p</code> ; <code>sp2</code> holds the <code>nullptr</code> .
<code>sp.~shared_ptr()</code>	Delete the pointer <code>sp</code> holds if <code>sp</code> is the last <code>shared_ptr</code> for that pointer.
<code>sp = sp2</code>	Copy assignment: if <code>sp</code> is the last shared pointer to refer to its pointer, delete that pointer; <code>sp</code> and <code>sp2</code> both hold <code>sp2</code> 's <code>p</code> .
<code>sp = move(sp2)</code>	Move assignment: if <code>sp</code> is the last shared pointer to refer to its pointer, delete that pointer; <code>sp</code> holds <code>sp2</code> 's <code>p</code> ; <code>sp2</code> holds the <code>nullptr</code> .
<code>p=sp.get()</code>	<code>p</code> is the pointer held by <code>sp</code> .
<code>n=sp.use_count()</code>	How many <code>shared_ptr</code> s refer to the pointer held by <code>sp</code> ?
<code>sp.reset(p)</code>	If <code>sp</code> is the last shared pointer to refer to its pointer, delete that pointer; <code>sp</code> holds <code>p</code> .
<code>sp=make_shared&lt;X&gt;(args)</code>	<code>sp</code> holds <code>new&lt;X&gt;(args)</code> .

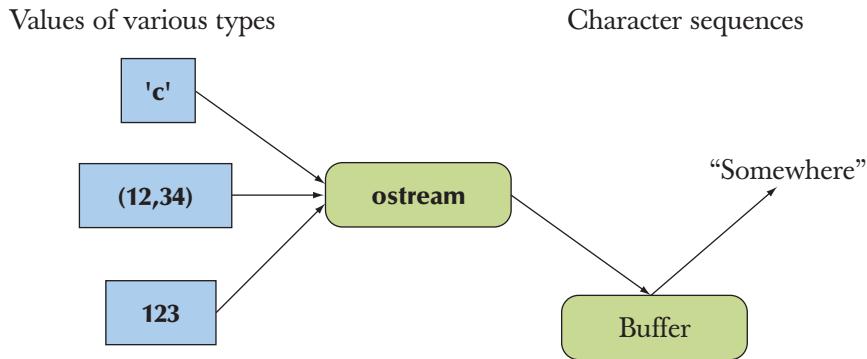
The usual pointer operations, such as `*`, `->`, `==`, and `<`, can be used for `shared_ptr`s. Additionally, a `shared_ptr` can be defined to use a delete action different from plain `delete`.

There is also a `weak_ptr` for breaking loops of `shared_ptr`s.

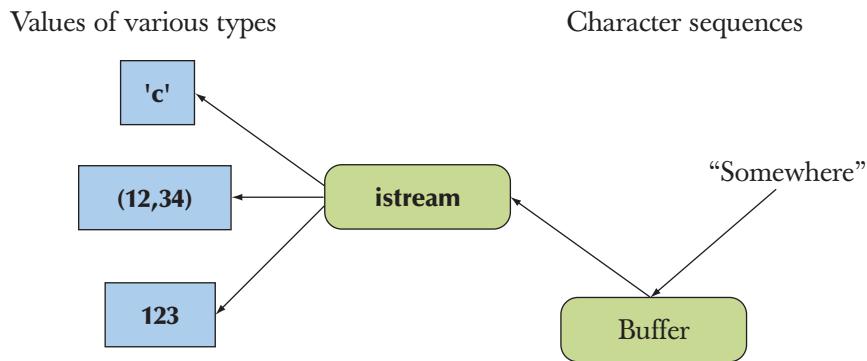
## B.7 I/O streams

The I/O stream library provides formatted and unformatted buffered I/O of text and numeric values. The definitions for I/O stream facilities are found in `<iostream>`, `<ostream>`, etc.; see §B.1.1.

An **ostream** converts typed objects to a stream of characters (bytes):



An **istream** converts a stream of characters (bytes) to typed objects:



An **iostream** is a stream that can act as both an **istream** and an **ostream**. The buffers in the diagrams are “stream buffers” (**streambufs**). Look them up in an expert-level textbook if you ever need to define a mapping from an **iostream** to a new kind of device, file, or memory.

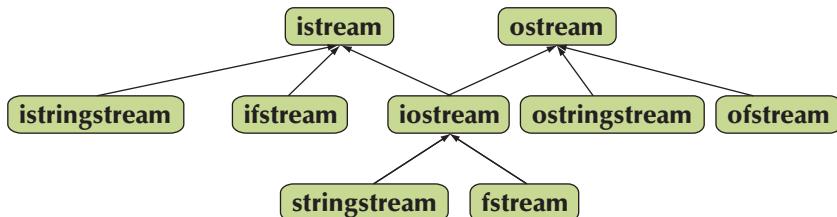
There are three standard streams:

#### Standard I/O streams

<b>cout</b>	the standard character output (often by default a screen)
<b>cin</b>	the standard character input (often by default a keyboard)
<b>cerr</b>	the standard character error output (unbuffered)

### B.7.1 I/O streams hierarchy

An **istream** can be connected to an input device (e.g., a keyboard), a file, or a **string**. Similarly, an **ostream** can be connected to an output device (e.g., a text window), a file, or a **string**. The I/O stream facilities are organized in a class hierarchy:



A stream can be opened either by a constructor or by an **open()** call:

Stream types	
<b>stringstream(m)</b>	Make an empty string stream with mode <b>m</b> .
<b>stringstream(s,m)</b>	Make a string stream containing <b>string s</b> with mode <b>m</b> .
<b>fstream()</b>	Make a file stream for later opening.
<b>fstream(s,m)</b>	Open a file called <b>s</b> with mode <b>m</b> and make a file stream to refer to it.
<b>fs.open(s,m)</b>	Open a file called <b>s</b> with mode <b>m</b> and have <b>fs</b> refer to it.
<b>fs.is_open()</b>	Is <b>fs</b> open?

For file streams, the file name is a C-style string.

You can open a file in one of several modes:

Stream modes	
<b>ios_base::app</b>	append (i.e., add to the end of the file)
<b>ios_base::ate</b>	"at end" (open and seek to the end)
<b>ios_base::binary</b>	binary mode — beware of system-specific behavior
<b>ios_base::in</b>	for reading
<b>ios_base::out</b>	for writing
<b>ios_base::trunc</b>	truncate the file to 0 length

In each case, the exact effect of opening a file may depend on the operating system, and if an operating system cannot honor a request to open a file in a certain way, the result will be a stream that is not in the **good()** state.

An example:

```
void my_code(ostream& os);           // my code can use any ostream

ostringstream os;                  // o for "output"
ofstream of("my_file");
if (!of.error("couldn't open 'my_file' for writing"));
my_code(os);                      // use a string
my_code(of);                     // use a file
```

See §11.3.

### B.7.2 Error handling

An **iostream** can be in one of four states:

Stream states	
<b>good()</b>	The operations succeeded.
<b>eof()</b>	We hit end of input ("end of file").
<b>fail()</b>	Something unexpected happened (e.g., we looked for a digit and found ' <b>x</b> ').
<b>bad()</b>	Something unexpected and serious happened (e.g., a disk read error).

By using **s.exceptions()**, a programmer can request an **iostream** to throw an exception if it turns from **good()** into another state (see §10.6).

Any operation attempted on a stream that is not in the **good()** state has no effect; it is a "no op."

An **iostream** can be used as a condition. In that case, the condition is true (succeeds) if the state of the **iostream** is **good()**. That is the basis for the common idiom for reading a stream of values:

```
for (X buf; cin>>buf; ) { // buf is an "input buffer" for holding one value of type X
    // . . . do something with buf . . .
}
// we get here when >> couldn't read another X from cin
```

### B.7.3 Input operations

Input operations are found in `<iostream>` except for the ones reading into a `string`; those are found in `<string>`:

Formatted input	
<code>in &gt;&gt; x</code>	Read from <code>in</code> into <code>x</code> according to <code>x</code> 's type.
<code>getline(in,s)</code>	Read a line from <code>in</code> into the string <code>s</code> .

Unless otherwise stated, an `istream` operation returns a reference to its `istream`, so that we can “chain” operations, for example, `cin>>x>>y;`.

Unformatted input	
<code>x=in.get()</code>	Read one character from <code>in</code> and return its integer value.
<code>in.get(c)</code>	Read a character from <code>in</code> into <code>c</code> .
<code>in.get(p,n)</code>	Read at most <code>n</code> characters from <code>in</code> into the array starting at <code>p</code> .
<code>in.get(p,n,t)</code>	Read at most <code>n</code> characters from <code>in</code> into the array starting at <code>p</code> ; consider <code>t</code> a terminator.
<code>in.getline(p,n)</code>	Read at most <code>n</code> characters from <code>in</code> into the array starting at <code>p</code> ; remove the terminator from <code>in</code> .
<code>in.getline(p,n,t)</code>	Read at most <code>n</code> characters from <code>in</code> into the array starting at <code>p</code> ; consider <code>t</code> a terminator; remove the terminator from <code>in</code> .
<code>in.read(p,n)</code>	Read at most <code>n</code> characters from <code>in</code> into the array starting at <code>p</code> .
<code>x=in.gcount()</code>	<code>x</code> is the number of characters read by the most recent unformatted input operation on <code>in</code> .
<code>in.unget()</code>	Back up the stream so that the next character read is the same as the previous read.
<code>in.putback(x)</code>	Put <code>x</code> “back” into the stream so that it will be the next character read.

The `get()` and `getline()` functions place a `0` at the end of the characters (if any) written to `p[0] . . . ; getline()` removes the terminator (`t`) from the input, if found, whereas `get()` does not. A `read(p,n)` does not write a `0` to the array after the characters read. Obviously, the formatted input operators are simpler to use and less error-prone than the unformatted ones.

### B.7.4 Output operations

Output operations are found in `<ostream>` except for the ones writing out a `string`; those are found in `<string>`:

Output operations	
<code>out &lt;&lt; x</code>	Write <code>x</code> to <code>out</code> according to <code>x</code> 's type.
<code>out.put(c)</code>	Write the character <code>c</code> to <code>out</code> .
<code>out.write(p,n)</code>	Write the characters <code>p[0]..p[n-1]</code> to <code>out</code> .

Unless otherwise stated, an `ostream` operation returns a reference to its `ostream`, so that we can “chain” operations, for example, `cout << x << y;`.

### B.7.5 Formatting

The format of stream I/O is controlled by a combination of object type, stream state, locale information (see `<locale>`), and explicit operations. Chapters 10 and 11 explain much of this. Here, we just list the standard manipulators (operations modifying the state of a stream) because they provide the most straightforward way of modifying formatting.

Locales are beyond the scope of this book.

### B.7.6 Standard manipulators

The standard library provides manipulators corresponding to the various format states and state changes. The standard manipulators are defined in `<iost>`, `<iostream>`, `<ostream>`, `<iostream>`, and `<iomanip>` (for manipulators that take arguments):

I/O manipulators	
<code>s&lt;&lt;boolalpha</code>	Use symbolic representation of <code>true</code> and <code>false</code> (input and output).
<code>s&lt;&lt;noboolalpha</code>	<code>s.unsetf(ios_base::boolalpha)</code> .
<code>s&lt;&lt;showbase</code>	On output prefix <code>oct</code> by <code>0</code> and <code>hex</code> by <code>0x</code> .
<code>s&lt;&lt;noshowbase</code>	<code>s.unsetf(ios_base::showbase)</code> .
<code>s&lt;&lt;showpoint</code>	Always show the decimal point.
<code>s&lt;&lt;noshowpoint</code>	<code>s.unsetf(ios_base::showpoint)</code> .
<code>s&lt;&lt;showpos</code>	Show <code>+</code> for positive numbers.

I/O manipulators ( <i>continued</i> )	
<code>s&lt;&lt;noshowpos</code>	<code>s.unsetf(ios_base::showpos).</code>
<code>s&gt;&gt;skipws</code>	Skip whitespace.
<code>s&gt;&gt;noskipws</code>	<code>s.unsetf(ios_base::skipws).</code>
<code>s&lt;&lt;uppercase</code>	Use upper case in numeric output, e.g., <code>1.2E10</code> and <code>0X1A2</code> rather than <code>1.2e10</code> and <code>0x1a2</code> .
<code>s&lt;&lt;nouppercase</code>	<code>x</code> and <code>e</code> rather than <code>X</code> and <code>E</code> .
<code>s&lt;&lt;internal</code>	Pad where marked in the formatting pattern.
<code>s&lt;&lt;left</code>	Pad after the value.
<code>s&lt;&lt;right</code>	Pad before the value.
<code>s&lt;&lt;dec</code>	Integer base is 10.
<code>s&lt;&lt;hex</code>	Integer base is 16.
<code>s&lt;&lt;oct</code>	Integer base is 8.
<code>s&lt;&lt;fixed</code>	Floating-point format dddd.dd.
<code>s&lt;&lt;scientific</code>	Scientific format d.ddddEdd.
<code>s&lt;&lt;defaultfloat</code>	Whatever format gives the most precise floating-point output.
<code>s&lt;&lt;endl</code>	Put ' <code>\n</code> ' and flush.
<code>s&lt;&lt;ends</code>	Put ' <code>\0</code> '.
<code>s&lt;&lt;flush</code>	Flush the stream.
<code>s&gt;&gt;ws</code>	Eat whitespace.
<code>s&lt;&lt;resetiosflags(f)</code>	Clear flags <code>f</code> .
<code>s&lt;&lt;setiosflags(f)</code>	Set flags <code>f</code> .
<code>s&lt;&lt;setbase(b)</code>	Output integers in base <code>b</code> .
<code>s&lt;&lt;setfill(c)</code>	Make <code>c</code> the fill character.
<code>s&lt;&lt;setprecision(n)</code>	Precision is <code>n</code> digits.
<code>s&lt;&lt;setw(n)</code>	Next field width is <code>n</code> characters.

Each of these operations returns a reference to its first (stream) operand, `s`. For example:

```
cout << 1234 << ',' << hex << 1234 << ',' << oct << 1234 << endl;
```

produces

`1234,4d2,2322`

and

```
cout << '(' << setw(4) << setfill('#') << 12 << ")" (" << 12 << ")\n";
```

produces

```
(##12) (12)
```

To explicitly set the general output format for floating-point numbers use

```
b.setf(ios_base::fmtflags(0), ios_base::floatfield)
```

See Chapter 11.

## B.8 String manipulation

The standard library offers character classification operations in `<cctype>`, strings with associated operations in `<string>`, regular expression matching in `<regex>`, and support for C-style strings in `<cstring>`.

### B.8.1 Character classification

The characters from the basic execution character set can be classified like this:

Character classification	
<b>isspace(c)</b>	Is <b>c</b> whitespace (' ', '\t', '\n', etc.)?
<b>isalpha(c)</b>	Is <b>c</b> a letter ('a'.. 'z', 'A'.. 'Z')? (Note: not '_').
<b>isdigit(c)</b>	Is <b>c</b> a decimal digit ('0'.. '9')?
<b>isxdigit(c)</b>	Is <b>c</b> a hexadecimal digit (decimal digit or 'a'.. 'f' or 'A'.. 'F')?
<b>isupper(c)</b>	Is <b>c</b> an uppercase letter?
<b>islower(c)</b>	Is <b>c</b> a lowercase letter?
<b>isalnum(c)</b>	Is <b>c</b> a letter or a decimal digit?
<b>iscntrl(c)</b>	Is <b>c</b> a control character (ASCII 0..31 and 127)?
<b>ispunct(c)</b>	Is <b>c</b> not a letter, digit, whitespace, or invisible control character?
<b>isprint(c)</b>	Is <b>c</b> printable (ASCII ' '.. '~')?
<b>isgraph(c)</b>	Is <b>isalpha(c)</b> or <b>isdigit(c)</b> or <b>ispunct(c)</b> ? (Note: not space.)

In addition, the standard library provides two useful functions for getting rid of case differences:

Upper and lower case	
<code>toupper(c)</code>	<code>c</code> or <code>c</code> 's uppercase equivalent
<code>tolower(c)</code>	<code>c</code> or <code>c</code> 's lowercase equivalent

Extended character sets, such as Unicode, are supported but are beyond the scope of this book.

## B.8.2 String

The standard library string class, `string`, is a specialization of a general string template `basic_string` for the character type `char`; that is, `string` is a sequence of `chars`:

String operations	
<code>s=s2</code>	Assign <code>s2</code> to <code>s</code> ; <code>s2</code> can be a string or a C-style string.
<code>s+=x</code>	Append <code>x</code> at end of <code>s</code> ; <code>x</code> can be a character, a string, or a C-style string.
<code>s[i]</code>	Subscripting.
<code>s+s2</code>	Concatenation; the result is a new string with the characters from <code>s</code> followed by the characters from <code>s2</code> .
<code>s==s2</code>	Comparison of string values; <code>s</code> or <code>s2</code> , but not both, can be a C-style string.
<code>s!=s2</code>	Comparison of string values; <code>s</code> or <code>s2</code> , but not both, can be a C-style string.
<code>s&lt;s2</code>	Lexicographical comparison of string values; <code>s</code> or <code>s2</code> , but not both, can be a C-style string.
<code>s&lt;=s2</code>	Lexicographical comparison of string values; <code>s</code> or <code>s2</code> , but not both, can be a C-style string.
<code>s&gt;s2</code>	Lexicographical comparison of string values; <code>s</code> or <code>s2</code> , but not both, can be a C-style string.
<code>s&gt;=s2</code>	Lexicographical comparison of string values; <code>s</code> or <code>s2</code> , but not both, can be a C-style string.
<code>s.size()</code>	Number of characters in <code>s</code> .
<code>s.length()</code>	Number of characters in <code>s</code> .

String operations ( <i>continued</i> )	
<code>s.c_str()</code>	C-style string version (zero terminated) of characters in <code>s</code> .
<code>s.begin()</code>	Iterator to the first character.
<code>s.end()</code>	Iterator to one beyond the end of <code>s</code> .
<code>s.insert(pos,x)</code>	Insert <code>x</code> before <code>s[pos]</code> ; <code>x</code> can be a <code>string</code> or a C-style string.
<code>s.append(x)</code>	Insert <code>x</code> after the last character of <code>s</code> ; <code>x</code> can be a <code>string</code> or a C-style string.
<code>s.erase(pos)</code>	Remove trailing characters from <code>s</code> starting with <code>s[pos]</code> . <code>s</code> 's size becomes <code>pos</code> .
<code>s.erase(pos,n)</code>	Remove <code>n</code> characters from <code>s</code> starting at <code>s[pos]</code> . <code>s</code> 's size becomes <code>max(pos,size-n)</code> .
<code>s.push_back(c)</code>	Append the character <code>c</code> .
<code>pos=s.find(x)</code>	Find <code>x</code> in <code>s</code> ; <code>x</code> can be a character, a string, or a C-style string; <code>pos</code> is the index of the first character found, or <code>string::npos</code> (a position off the end of <code>s</code> ).
<code>in&gt;&gt;s</code>	Read a word into <code>s</code> from <code>in</code> .

### B.8.3 Regular expression matching

The regular expression facilities are found in `<regex>`. The main functions are

- *Searching* for a string that matches a regular expression in an (arbitrarily long) stream of data – supported by `regex_search()`
- *Matching* a regular expression against a string (of known size) – supported by `regex_match()`
- *Replacement* of matches – supported by `regex_replace()`; not described in this book; see an expert-level text or manual

The result of a `regex_search()` or a `regex_match()` is a collection of matches, typically represented as an `smatch`:

```
regex row("^\w+ (\d+) (\d+) (\d+$); // data line
while (getline(in,line)) { // check data line
    smatch matches;
    if (!regex_match(line, matches, row))
        error("bad line", lineno);
```

```
// check row:  
int field1 = from_string<int>(matches[1]);  
int field2 = from_string<int>(matches[2]);  
int field3 = from_string<int>(matches[3]);  
// ...  
}
```

The syntax of regular expressions is based on characters with special meaning (Chapter 23):

Regular expression special characters	
.	any single character (a “wildcard”)
[	character class
{	count
(	begin grouping
)	end grouping
\	next character has a special meaning
*	zero or more
+	one or more
?	optional (zero or one)
	alternative (or)
^	start of line; negation
\$	end of line

Repetition	
{ n }	exactly <b>n</b> times
{ n, }	<b>n</b> or more times
{n,m}	at least <b>n</b> and at most <b>m</b> times
*	zero or more, that is, {0,}
+	one or more, that is, {1,}
?	optional (zero or one), that is {0,1}

Character classes	
<b>alnum</b>	any alphanumeric character or the underscore
<b>alpha</b>	any alphabetic character
<b>blank</b>	any whitespace character that is not a line separator
<b>ctrl</b>	any control character
<b>d</b>	any decimal digit
<b>digit</b>	any decimal digit
<b>graph</b>	any graphical character
<b>lower</b>	any lowercase character
<b>print</b>	any printable character
<b>punct</b>	any punctuation character
<b>s</b>	any whitespace character
<b>space</b>	any whitespace character
<b>upper</b>	any uppercase character
<b>w</b>	any word character (alphanumeric characters)
<b>xdigit</b>	any hexadecimal digit character

Several character classes are supported by shorthand notation:

Character class abbreviations		
<b>\d</b>	a decimal digit	<code>[:digit:]</code>
<b>\l</b>	a lowercase character	<code>[:lower:]</code>
<b>\s</b>	a space (space, tab, etc.)	<code>[:space:]</code>
<b>\u</b>	an uppercase character	<code>[:upper:]</code>
<b>\w</b>	a letter, a decimal digit, or an underscore (_)	<code>[:alnum:]</code>
<b>\D</b>	not \d	<code>^[:digit:]</code>
<b>\L</b>	not \l	<code>^[:lower:]</code>
<b>\S</b>	not \s	<code>^[:space:]</code>
<b>\U</b>	not \u	<code>^[:upper:]</code>
<b>\W</b>	not \w	<code>^[:alnum:]</code>

## B.9 Numerics

The C++ standard library provides the most basic building blocks for mathematical (scientific, engineering, etc.) calculations.

### B.9.1 Numerical limits

Each C++ implementation specifies properties of the built-in types, so that programmers can use those properties to check against limits, set sentinels, etc.

From `<limits>`, we get `numeric_limits<T>` for each built-in or library type `T`. In addition, a programmer can define `numeric_limits<X>` for a user-defined numeric type `X`. For example:

```
class numeric_limits<float> {
public:
    static const bool is_specialized = true;

    static constexpr int radix = 2;      // base of exponent (in this case, binary)
    static constexpr int digits = 24;    // number of radix digits in mantissa
    static constexpr int digits10 = 6;   // number of base-10 digits in mantissa

    static constexpr bool is_signed = true;
    static constexpr bool is_integer = false;
    static constexpr bool is_exact = false;

    static constexpr float min() { return 1.17549435E-38F; }    // example value
    static constexpr float max() { return 3.40282347E+38F; }    // example value
    static constexpr float lowest() { return -3.40282347E+38F; } // example value

    static constexpr float epsilon() { return 1.19209290E-07F; } // example value
    static constexpr float round_error() { return 0.5F; }           // example value

    static constexpr float infinity() { return /* some value */; }
    static constexpr float quiet_NaN() { return /* some value */; }
    static constexpr float signaling_NaN() { return /* some value */; }
    static constexpr float denorm_min() { return min(); }

    static constexpr int min_exponent = -125;                  // example value
    static constexpr int min_exponent10 = -37;                 // example value
    static constexpr int max_exponent = +128;                  // example value
    static constexpr int max_exponent10 = +38;                 // example value
```

```

static constexpr bool has_infinity = true;
static constexpr bool has_quiet_NaN = true;
static constexpr bool has_signaling_NaN = true;
static constexpr float_denorm_style has_denorm = denorm_absent;
static constexpr bool has_denorm_loss = false;

static constexpr bool is_iec559 = true;           // conforms to IEC-559
static constexpr bool is_bounded = true;
static constexpr bool is_modulo = false;
static constexpr bool traps = true;
static constexpr bool tininess_before = true;

static constexpr float_round_style round_style = round_to_nearest;
};

```

From `<limits.h>` and `<float.h>`, we get macros specifying key properties of integers and floating-point numbers, including:

Limit macros	
<b>CHAR_BIT</b>	number of bits in a <b>char</b> (usually 8)
<b>CHAR_MIN</b>	minimum <b>char</b> value
<b>CHAR_MAX</b>	maximum <b>char</b> value (usually 127 if <b>char</b> is signed and 255 if <b>char</b> is unsigned)

### B.9.2 Standard mathematical functions

The standard library provides the most common mathematical functions (defined in `<cmath>` and `<complex>`):

Standard mathematical functions	
<b>abs(x)</b>	absolute value
<b>ceil(x)</b>	smallest integer <b>&gt;= x</b>
<b>floor(x)</b>	largest integer <b>&lt;= x</b>
<b>round(x)</b>	round to the nearest integer (.5 rounds away from zero)
<b>sqrt(x)</b>	square root; <b>x</b> must be nonnegative
<b>cos(x)</b>	cosine

Standard mathematical functions ( <i>continued</i> )	
<b>sin(x)</b>	sine
<b>tan(x)</b>	tangent
<b>acos(x)</b>	arccosine; the result is nonnegative
<b>asin(x)</b>	arcsine; the result nearest to 0 is returned
<b>atan(x)</b>	arctangent
<b>sinh(x)</b>	hyperbolic sine
<b>cosh(x)</b>	hyperbolic cosine
<b>tanh(x)</b>	hyperbolic tangent
<b>exp(x)</b>	base-e exponential
<b>log(x)</b>	natural logarithm, base-e; <b>x</b> must be positive
<b>log10(x)</b>	base-10 logarithm

There are versions taking **float**, **double**, **long double**, and **complex** arguments. For each function, the return type is the same as the argument type.

If a standard mathematical function cannot produce a mathematically valid result, it sets the variable **errno**.

### B.9.3 Complex

The standard library provides complex number types **complex<float>**, **complex<double>**, and **complex<long double>**. A **complex<Scalar>** where **Scalar** is some other type supporting the usual arithmetic operations usually works but is not guaranteed to be portable.

```
template<class Scalar> class complex {
    // a complex is a pair of scalar values, basically a coordinate pair
    Scalar re, im;
public:
    constexpr complex(const Scalar & r, const Scalar & i) :re{r}, im{i} {}
    constexpr complex(const Scalar & r) :re{r}, im{Scalar{} } {}
    constexpr complex() :re{Scalar{} }, im{Scalar{} } {}

    Scalar real() { return re; }      // real part
    Scalar imag() { return im; }     // imaginary part

    // operators: = += -= *= ==
};
```

In addition to the members of `complex`, `<complex>` offers a host of useful operations:

Complex operators	
<code>z1+z2</code>	addition
<code>z1-z2</code>	subtraction
<code>z1*z2</code>	multiplication
<code>z1/z2</code>	division
<code>z1==z2</code>	equality
<code>z1!=z2</code>	inequality
<code>norm(z)</code>	the square of <code>abs(z)</code>
<code>conj(z)</code>	conjugate: if <code>z</code> is <code>{re,im}</code> then <code>conj(z)</code> is <code>{re,-im}</code>
<code>polar(x,y)</code>	make a complex given polar coordinates (rho,theta)
<code>real(z)</code>	real part
<code>imag(z)</code>	imaginary part
<code>abs(z)</code>	also known as rho
<code>arg(z)</code>	also known as theta
<code>out &lt;&lt; z</code>	complex output
<code>in &gt;&gt; z</code>	complex input

The standard mathematical functions (see §B.9.2) are also available for complex numbers. Note: `complex` does not provide `<` or `%`; see also §24.9.

### B.9.4 `valarray`

The standard `valarray` is a single-dimensional numerical array; that is, it provides arithmetic operations for an array type (much like `Matrix` in Chapter 24) plus support for slices and strides.

### B.9.5 Generalized numerical algorithms

These algorithms from `<numeric>` provide general versions of common operations on sequences of numerical values:

Numerical algorithms	
<code>x = accumulate(b,e,i)</code>	<code>x</code> is the sum of <code>i</code> and the elements of <code>[b:e)</code> .
<code>x = accumulate(b,e,i,f)</code>	Accumulate, but with <code>f</code> instead of <code>+</code> .

Numerical algorithms ( <i>continued</i> )	
<b>x = inner_product(b,e,b2,i)</b>	<b>x</b> is the inner product of <b>[b:e]</b> and <b>[b2:b2+(e-b)]</b> , that is, the sum of <b>i</b> and <b>(*p1)*(*p2)</b> for all <b>p1</b> in <b>[b:e]</b> and all corresponding <b>p2</b> in <b>[b2:b2+(e-b)]</b> .
<b>x = inner_product(b,e,b2,i,f,f2)</b>	<b>inner_product</b> , but with <b>f</b> and <b>f2</b> instead of <b>+</b> and <b>*</b> , respectively.
<b>p=partial_sum(b,e,out)</b>	Element <b>i</b> of <b>[out:p]</b> is the sum of elements <b>0..i</b> of <b>[b:e]</b> .
<b>p=partial_sum(b,e,out,f)</b>	<b>partial_sum</b> , using <b>f</b> instead of <b>+</b> .
<b>p=adjacent_difference(b,e,out)</b>	Element <b>i</b> of <b>[out:p]</b> is <b>*(b+i)-*(b+i-1)</b> for <b>i&gt;0</b> ; if <b>e-b&gt;0</b> then <b>*out</b> is <b>*b</b> .
<b>p=adjacent_difference(b,e,out,f)</b>	<b>adjacent_difference</b> , using <b>f</b> instead of <b>-</b> .
<b>iota(b,e,v)</b>	For each element of <b>[b:e]</b> assign <b>++v</b> .

For example:

```
vector<int> v(100);
iota(v.begin(),v.end(),0);      // v=={1, 2,3,4,5 . . . 100}
```

## B.9.6 Random numbers

In **<random>**, the standard library provides random number engines and distributions (§24.7). By default use the **default\_random\_engine**, which is chosen for wide applicability and low cost.

Distributions include:

Distributions	
<b>uniform_int_distribution&lt;int&gt; {low, high}</b>	value in <b>[low:high]</b>
<b>uniform_real_distribution&lt;int&gt;{low,high}</b>	value in <b>[low:high]</b>
<b>exponential_distribution&lt;double&gt;{lambda}</b>	value in <b>[0:∞)</b>
<b>bernoulli_distribution{p}</b>	value in <b>[true:false]</b>
<b>normal_distribution&lt;double&gt;{median,spread}</b>	value in <b>(-∞:∞)</b>

A distribution can be called with an engine as its argument. For example:

```
uniform_real_distribution<> dist;
default_random_engine engn;
for (int i = 0; i<10; ++i)
    cout << dist(engn) << ' ';
```

## B.10 Time

In **<chrono>**, the standard library provides facilities for timing. A clock counts time in number of clock ticks and reports the current point in time as the result of a call of **now()**. Three clocks are defined:

- **system\_clock**: the default system clock
- **steady\_clock**: a clock, **c**, for which **c.now()<=c.now()** for consecutive calls of **now()** and the time between clock ticks is constant
- **high\_resolution\_clock**: the highest-resolution clock available on a system

A number of clock ticks for a given clock is converted into a conventional unit of time, such as **seconds**, **milliseconds**, and **nanoseconds**, by the function **duration\_cast<>()**. For example:

```
auto t = steady_clock::now();
// . . . do something . .
auto d = steady_clock::now()-t;      // something took d time units

cout << "something took "
     << duration_cast<milliseconds>(d).count() << "ms";
```

This will print the time that “something” took in milliseconds. See also §26.6.1.

## B.11 C standard library functions

The standard library for the C language is with very minor modifications incorporated into the C++ standard library. The C standard library provides quite a few functions that have proved useful over the years in a wide variety of contexts – especially for relatively low-level programming. Here, we have organized them into a few conventional categories:

- C-style I/O
- C-style strings

- Memory
- Date and time
- Etc.

There are more C standard library functions than we present here; see a good C textbook, such as Kernighan and Ritchie, *The C Programming Language* (K&R), if you need to know more.

### B.11.1 Files

The `<stdio>` I/O system is based on “files.” A file (a `FILE*`) can refer to a file or to one of the standard input and output streams, `stdin`, `stdout`, and `stderr`. The standard streams are available by default; other files need to be opened:

File open and close	
<code>f=fopen(s,m)</code>	Open a file stream for a file named <code>s</code> with the mode <code>m</code> .
<code>x=fclose(f)</code>	Close file stream <code>f</code> ; return <code>0</code> if successful.

A “mode” is a string containing one or more directives specifying how a file is to be opened:

File modes	
<code>"r"</code>	reading
<code>"w"</code>	writing (discard previous contents)
<code>"a"</code>	append (add at end)
<code>"r+"</code>	reading and writing
<code>"w+"</code>	reading and writing (discard previous contents)
<code>"b"</code>	binary; use together with one or more other modes

There may be (and usually are) more options on a specific system. Some options can be combined; for example, `fopen("foo","rb")` tries to open a file called `foo` for binary reading. The I/O modes should be the same for stdio and `iostreams` (§B.7.1).

### B.11.2 The `printf()` family

The most popular C standard library functions are the I/O functions. However, we recommend `iostreams` because that library is type-safe and extensible. The

formatted output function, **printf()**, is widely used (also in C++ programs) and widely imitated in other programming languages:

<b>printf</b>	
<b>n=printf(fmt,args)</b>	Print the “format string” <b>fmt</b> to <b>stdout</b> , inserting the arguments <b>args</b> as appropriate.
<b>n=fprintf(f,fmt,args)</b>	Print the “format string” <b>fmt</b> to file <b>f</b> , inserting the arguments <b>args</b> as appropriate.
<b>n=sprintf(s,fmt,args)</b>	Print the “format string” <b>fmt</b> to the C-style string <b>s</b> , inserting the arguments <b>args</b> as appropriate.

For each version, **n** is the number of characters written or a negative number if the output failed. The return value from **printf()** is essentially always ignored.

The declaration of **printf()** is

```
int printf(const char* format . . .);
```

In other words, it takes a C-style string (typically a string literal) followed by an arbitrary number of arguments of arbitrary type. The meaning of those “extra arguments” is controlled by conversion specifications, such as **%c** (print as character) and **%d** (print as decimal integer), in the format string. For example:

```
int x = 5;
const char* p = "asdf";
printf("the value of x is '%d' and the value of p is '%s'\n",x,p);
```

A character following a **%** controls the handling of an argument. The first **%** applies to the first “extra argument” (here, **%d** applies to **x**), the second **%** to the second “extra argument” (here, **%s** applies to **p**), and so on. In particular, the output of that call to **printf()** is

**the value of x is '5' and the value of p is 'asdf'**

followed by a newline.

In general, the correspondence between a **%** conversion directive and the type to which it is applied cannot be checked, and when it can, it usually is not. For example:

```
printf("the value of x is '%s' and the value of p is '%d'\n",x,p); // oops
```

The set of conversion specifications is quite large and provides a great degree of flexibility (and possibilities for confusion). Following the %, there may be:

- an optional minus sign that specifies left adjustment of the converted value in the field.
  - + an optional plus sign that specifies that a value of a signed type will always begin with a + or – sign.
  - 0 an optional zero that specifies that leading zeros are used for padding of a numeric value. If – or a precision is specified, this 0 is ignored.
  - # an optional # that specifies that floating-point values will be printed with a decimal point even if no nonzero digits follow, that trailing zeros will be printed, that octal values will be printed with an initial 0, and that hexadecimal values will be printed with an initial 0x or 0X.
  - d an optional digit string specifying a field width; if the converted value has fewer characters than the field width, it will be blank-padded on the left (or right, if the left-adjustment indicator has been given) to make up the field width; if the field width begins with a zero, zero padding will be done instead of blank padding.
  - . an optional period that serves to separate the field width from the next digit string.
- dd an optional digit string specifying a precision that specifies the number of digits to appear after the decimal point, for e- and f-conversion, or the maximum number of characters to be printed from a string.
- \* a field width or precision may be \* instead of a digit string. In this case, an integer argument supplies the field width or precision.
- h an optional character h, specifying that a following d, o, x, or u corresponds to a short integer argument.
- l an optional character l (the letter l), specifying that a following d, o, x, or u corresponds to a long integer argument.
- L an optional character L, specifying that a following e, E, g, G, or f corresponds to a long double argument.
- % indicating that the character % is to be printed; no argument is used.
- c a character that indicates the type of conversion to be applied. The conversion characters and their meanings are:
  - d The integer argument is converted to decimal notation.
  - i The integer argument is converted to decimal notation.
  - o The integer argument is converted to octal notation.
  - x The integer argument is converted to hexadecimal notation.

- X** The integer argument is converted to hexadecimal notation.
- f** The **float** or **double** argument is converted to decimal notation in the style **[−]ddd.ddd**. The number of *ds* after the decimal point is equal to the precision for the argument. If necessary, the number is rounded. If the precision is missing, six digits are given; if the precision is explicitly **0** and **#** isn't specified, no decimal point is printed.
- e** The **float** or **double** argument is converted to decimal notation in the scientific style **[−]d.ddde+dd** or **[−]d.ddde-dd**, where there is one digit before the decimal point and the number of digits after the decimal point is equal to the precision specification for the argument. If necessary, the number is rounded. If the precision is missing, six digits are given; if the precision is explicitly **0** and **#** isn't specified, no digits and no decimal point are printed.
- E** Like **e**, but with an uppercase **E** used to identify the exponent.
- g** The **float** or **double** argument is printed in style **d**, in style **f**, or in style **e**, whichever gives the greatest precision in minimum space.
- G** Like **g**, but with an uppercase **E** used to identify the exponent.
- c** The character argument is printed. Null characters are ignored.
- s** The argument is taken to be a string (character pointer), and characters from the string are printed until a null character or until the number of characters indicated by the precision specification is reached; however, if the precision is **0** or missing, all characters up to a null are printed.
- p** The argument is taken to be a pointer. The representation printed is implementation dependent.
- u** The unsigned integer argument is converted to decimal notation.
- n** The number of characters written so far by the call of **printf()**, **fprintf()**, or **sprintf()** is written to the **int** pointed to by the pointer-to-**int** argument.

In no case does a nonexistent or small field width cause truncation of a field; padding takes place only if the specified field width exceeds the actual width.

Because C does not have user-defined types in the sense that C++ has, there are no provisions for defining output formats for user-defined types, such as **complex**, **vector**, or **string**.

The C standard output, **stdout**, corresponds to **cout**. The C standard input, **stdin**, corresponds to **cin**. The C standard error output, **stderr**, corresponds to **cerr**. This correspondence between C standard I/O and C++ I/O streams is so

close that C-style I/O and I/O streams can share a buffer. For example, a mix of **cout** and **stdout** operations can be used to produce a single output stream (that's not uncommon in mixed C and C++ code). This flexibility carries a cost. For better performance, don't mix stdio and **iostream** operations for a single stream and call **ios\_base::sync\_with\_stdio(false)** before the first I/O operation.

The stdio library provides a function, **scanf()**, that is an input operation with a style that mimics **printf()**. For example:

```
int x;
char s[buf_size];
int i = scanf("the value of x is %d and the value of s is %s\n", &x, s);
```

Here, **scanf()** tries to read an integer into **x** and a sequence of non-whitespace characters into **s**. Non-format characters specify that the input should contain that character. For example,

**the value of x is '123' and the value of s is 'string '\n'**

will read **123** into **x** and **string** followed by a **\n** into **s**. If the call of **scanf()** succeeds, the result value (**i** in the call above) will be the number of argument pointers assigned to (hopefully 2 in the example); otherwise, **EOF**. This way of specifying input is error-prone (e.g., what would happen if you forgot the space after **string** on that input line?). All arguments to **scanf()** must be pointers. We strongly recommend against the use of **scanf()**.

So what can we do for input if we are obliged to use stdio? One popular answer is, “Use the standard library function **gets()**”:

```
// very dangerous code:
char s[buf_size];
char* p = gets(s); // read a line into s
```

The call **p=gets(s)** reads characters into **s** until a newline or an end of file is encountered and a **\0** character is placed after the last character written to **s**. If an end of file is encountered or if an error occurred, **p** is set to **NULL** (that is, **0**); otherwise it is set to **s**. Never use **gets(s)** or its rough equivalent (**scanf("%s",s)**)! For years, they were the favorites of virus writers: by providing an input that overflows the input buffer (**s** in the example), a program can be corrupted and a computer potentially taken over by an attacker. The **sprintf()** function suffers similar buffer-overflow problems.

The stdio library also provides simple and useful character read and write functions:

stdio character functions	
<b>x=getc(st)</b>	Read a character from input stream <b>st</b> ; return the character's integer value; <b>x==EOF</b> if end of file or an error occurred.
<b>x=putc(c,st)</b>	Write the character <b>c</b> to the output stream <b>st</b> ; return the integer value of the character written; <b>x==EOF</b> if an error occurred.
<b>x=getchar()</b>	Read a character from <b>stdin</b> ; return the character's integer value; <b>x==EOF</b> if end of file or an error occurred.
<b>x=putchar(c)</b>	Write the character <b>c</b> to <b>stdout</b> ; return the integer value of the character written; <b>x==EOF</b> if an error occurred.
<b>x=ungetc(c,st)</b>	Put <b>c</b> back onto the input stream <b>st</b> ; return the integer value of the character pushed; <b>x==EOF</b> if an error occurred.

Note that the result of these functions is an **int** (not a **char**, or **EOF** couldn't be returned). For example, this is a typical C-style input loop:

```
int ch; /* not char ch; */
while ((ch=getchar())!=EOF) { /* do something */ }
```

Don't do two consecutive **ungetc()**s on a stream. The result of that is undefined and (therefore) non-portable.

There are more stdio functions; see a good C textbook, such as K&R, if you need to know more.

### B.11.3 C-style strings

A C-style string is a zero-terminated array of **chars**. This notion of a string is supported by a set of functions defined in **<cstring>** (or **<string.h>**; note: *not <string>*) and **<cstdlib>**. These functions operate on C-style strings through **char\*** pointers (**const char\*** pointers for memory that's only read):

C-style string operations	
<b>x=strlen(s)</b>	Count the characters (excluding the terminating 0).
<b>p=strcpy(s,s2)</b>	Copy <b>s2</b> into <b>s</b> ; [ <b>s:s+n</b> ] and [ <b>s2:s2+n</b> ] may not overlap; <b>p=s</b> ; the terminating 0 is copied.
<b>p=strcat(s,s2)</b>	Copy <b>s2</b> onto the end of <b>s</b> ; <b>p=s</b> ; the terminating 0 is copied.
<b>x=strcmp(s,s2)</b>	Compare lexicographically: if <b>s&lt;s2</b> then <b>x</b> is negative; if <b>s==s2</b> then <b>x==0</b> ; if <b>s&gt;s2</b> then <b>x</b> is positive.

### C-style string operations (*continued*)

<b>p=strncpy(s,s2,n)</b>	<b>strcpy</b> ; max <b>n</b> characters; may fail to copy terminating 0; <b>p=s</b> .
<b>p=strncat(s,s2,n)</b>	<b>strcat</b> ; max <b>n</b> characters; may fail to copy terminating 0; <b>p=s</b> .
<b>x=strncmp(s,s2,n)</b>	<b>strcmp</b> ; max <b>n</b> characters.
<b>p=strchr(s,c)</b>	Make <b>p</b> point to the first <b>c</b> in <b>s</b> .
<b>p=strrchr(s,c)</b>	Make <b>p</b> point to the last <b>c</b> in <b>s</b> .
<b>p=strstr(s,s2)</b>	Make <b>p</b> point to the first character of <b>s</b> that starts a substring equal to <b>s2</b> .
<b>p=strpbrk(s,s2)</b>	Make <b>p</b> point to the first character of <b>s</b> also found in <b>s2</b> .
<b>x=atof(s)</b>	Extract a <b>double</b> from <b>s</b> .
<b>x=atoi(s)</b>	Extract an <b>int</b> from <b>s</b> .
<b>x=atol(s)</b>	Extract a <b>long int</b> from <b>s</b> .
<b>x=strtod(s,p)</b>	Extract a <b>double</b> from <b>s</b> ; set <b>p</b> to the first character after the <b>double</b> .
<b>x=strtol(s,p)</b>	Extract a <b>long int</b> from <b>s</b> ; set <b>p</b> to the first character after the <b>long</b> .
<b>x=strtoul(s,p)</b>	Extract an <b>unsigned long int</b> from <b>s</b> ; set <b>p</b> to the first character after the <b>long</b> .

Note that in C++, **strchr()** and **strstr()** are duplicated to make them type-safe (they can't turn a **const char\*** into a **char\*** the way the C equivalents can); see also §27.5.

An extraction function looks into its C-style string argument for a conventionally formatted representation of a number, such as "**124**" and "**1.4**". If no such representation is found, the extraction function returns **0**. For example:

```
int x = atoi("fortytwo");      /* x becomes 0 */
```

### B.11.4 Memory

The memory manipulation functions operate on “raw memory” (no type known) through **void\*** pointers (**const void\*** pointers for memory that's only read):

#### C-style memory operations

<b>q=memcpy(p, p2, n)</b>	Copy <b>n</b> bytes from <b>p2</b> to <b>p</b> (like <b>strcpy</b> ); <b>[p:p+n]</b> and <b>[p2:p2+n]</b> may not overlap; <b>q=p</b> .
<b>q=memmove(p,p2,n)</b>	Copy <b>n</b> bytes from <b>p2</b> to <b>p</b> ; <b>q=p</b> .
<b>x=memcmp(p,p2,n)</b>	Compare <b>n</b> bytes from <b>p2</b> to the equivalent <b>n</b> bytes from <b>p</b> (like <b>strcmp</b> ).

C-style memory operations ( <i>continued</i> )	
<b>q=memchr(p,c,n)</b>	Find <b>c</b> (converted to an <b>unsigned char</b> ) in <b>p[0]..p[n-1]</b> and let <b>q</b> point to that element; <b>q=0</b> if <b>c</b> is not found.
<b>q=memset(p,c,n)</b>	Copy <b>c</b> (converted to an <b>unsigned char</b> ) into each of <b>p[0]..[n-1]; q=p</b> .
<b>p=calloc(n,s)</b>	Allocate <b>n*s</b> bytes initialized to <b>0</b> on the free store; <b>p=0</b> if <b>n*s</b> bytes could not be allocated.
<b>p=malloc(s)</b>	Allocate <b>s</b> uninitialized bytes on the free store; <b>p=0</b> if <b>s</b> bytes could not be allocated.
<b>q=realloc(p,s)</b>	Allocate <b>s</b> bytes on the free store; <b>p</b> must be a pointer returned by <b>malloc()</b> or <b>calloc()</b> ; if possible reuse the space pointed to by <b>p</b> ; if that is not possible copy all bytes in the area pointed to by <b>p</b> to a new area; <b>q=0</b> if <b>s</b> bytes could not be allocated.
<b>free(p)</b>	Deallocate the memory pointed to by <b>p</b> ; <b>p</b> must be a pointer returned by <b>malloc()</b> , <b>calloc()</b> , or <b>realloc()</b> .

Note that **malloc()**, etc. do not invoke constructors and **free()** doesn't invoke destructors. Do not use these functions for types with constructors or destructors. Also, **memset()** should never be used for any type with a constructor.

The **mem\*** functions are found in **<cstring>** and the allocation functions in **<cstdlib>**.

See also §27.5.2.

### B.11.5 Date and time

In **<ctime>**, you can find several types and functions related to date and time:

Date and time types	
<b>clock_t</b>	an arithmetic type for holding short time intervals (maybe just intervals of a few minutes)
<b>time_t</b>	an arithmetic type for holding long time intervals (maybe centuries)
<b>tm</b>	a <b>struct</b> for holding date and time (since year 1900)

**struct tm** is defined like this:

```
struct tm {
    int tm_sec;   // second of minute [0:61]; 60 and 61 represent leap seconds
    int tm_min;   // minute of hour [0,59]
```

```

int tm_hour; // hour of day [0,23]
int tm_mday; // day of month [1,31]
int tm_mon; // month of year [0,11]; 0 means January (note: not [1:12])
int tm_year; // year since 1900; 0 means year 1900, and 102 means 2002
int tm_wday; // days since Sunday [0,6]; 0 means Sunday
int tm_yday; // days since January 1 [0,365]; 0 means January 1
int tm_isdst; // hours of Daylight Savings Time
};


```

Date and time functions:

```

clock_t clock(); // number of clock ticks since the start of the program

time_t time(time_t* pt); // current calendar time
double difftime(time_t t2, time_t t1); // t2-t1 in seconds

tm* localtime(const time_t* pt); // local time for the *pt
tm* gmtime(const time_t* pt); // Greenwich Mean Time (GMT) tm for *pt, or 0

time_t mktime(tm* ptm); // time_t for *ptm, or time_t(-1)

char* asctime(const tm* ptm); // C-style string representation for *ptm
char* ctime(const time_t* t) { return asctime(localtime(t)); }

```

An example of the result of a call of **asctime()** is "Sun Sep 16 01:03:52 1973\n".

An amazing zoo of formatting options for **tm** is provided by a function called **strftime()**. Look it up if you need it.

## B.10.6 Etc.

In **<cstdlib>** we find:

Etc. <b>stdlib</b> functions	
<b>abort()</b>	Terminate the program “abnormally.”
<b>exit(n)</b>	Terminate the program with value <b>n</b> ; <b>n==0</b> means successful termination.
<b>system(s)</b>	Execute the C-style string as a command (system dependent).
<b>qsort(b,n,s,cmp)</b>	Sort the array starting at <b>b</b> with <b>n</b> elements of size <b>s</b> using the comparison function <b>cmp</b> .
<b>bsearch(k,b,n,s,cmp)</b>	Search for <b>k</b> in the sorted array starting at <b>b</b> with <b>n</b> elements of size <b>s</b> using the comparison function <b>cmp</b> .

The comparison function (**cmp**) used by **qsort()** and **bsearch()** must have the type

```
int (*cmp)(const void* p, const void* q);
```

That is, no type information is known to the sort function that simply “sees” its array as a sequence of bytes. The integer returned is

- Negative if **\*p** is considered less than **\*q**
- Zero if **\*p** is considered equal to **\*q**
- Positive if **\*p** is considered greater than **\*q**

Note that **exit()** and **abort()** do not invoke destructors. If you want destructors called for constructed automatic and static objects (§A.4.2), throw an exception.

For more standard library functions see K&R or some other reputable C language reference.

## B.12 Other libraries

Looking through the standard library facilities, you’ll undoubtedly have failed to find something you could use. Compared to the challenges faced by programmers and the number of libraries available in the world, the C++ standard library is minute. There are many libraries for

- Graphical user interfaces
- Advanced math
- Database access
- Networking
- XML
- Date and time
- File system manipulation
- 3D graphics
- Animation
- Etc.

However, such libraries are not part of the standard. You can find them by searching the web or by asking friends and colleagues. Please don’t get the idea that the only useful libraries are those that are part of the standard library.





# Getting Started with Visual Studio

“The universe is not only queerer than we imagine, it’s queerer than we *can* imagine.”

—J. B. S. Haldane

This appendix explains the steps you have to go through to enter a program, compile it, and have it run using Microsoft Visual Studio.

### C.1 Getting a program to run

### C.2 Installing Visual Studio

### C.3 Creating and running a program

#### C.3.1 Create a new project

#### C.3.2 Use the `std_lib_facilities.h` header file

#### C.3.3 Add a C++ source file to the project

#### C.3.4 Enter your source code

#### C.3.5 Build an executable program

#### C.3.6 Execute the program

#### C.3.7 Save the program

### C.4 Later

## C.1 Getting a program to run

To get a program to run, you need to somehow place the files together (so that when a file refers to another – e.g., your source file refers to a header file – it finds it). You then need to invoke the compiler and the linker (if nothing else, then to link to the C++ standard library), and finally you need to run (execute) the program. There are several ways of doing that, and different systems (e.g., Windows and Linux) have different conventions and tool sets. However, you can run all of the examples from this book on all major systems using any of the major tool sets. This appendix explains how to do it for one popular system, Microsoft’s Visual Studio.

Personally, we find few exercises as frustrating as getting a first program to work on a new and strange system. This is a task for which it makes sense to ask for help. However, if you do get help, be sure that the helper teaches you how to do it, rather than just doing it for you.

## C.2 Installing Visual Studio

Visual Studio is an interactive development environment (IDE) for Windows. If Visual Studio is not installed on your computer, you may purchase a copy and follow the instructions that come with it, or download and install the free Visual C++ Express from [www.microsoft.com/express/download](http://www.microsoft.com/express/download). The description here is based on Visual Studio 2005. Other versions may differ slightly.

## C.3 Creating and running a program

The steps are:

1. Create a new project.
2. Add a C++ source file to the project.
3. Enter your source code.
4. Build an executable file.
5. Execute the program.
6. Save the program.

### C.3.1 Create a new project

In Visual Studio, a “project” is a collection of files that together provide what it takes to create and run a program (also called an *application*) under Windows.

1. Open the Visual C++ IDE by clicking the Microsoft Visual Studio 2005 icon, or select it from **Start > Programs > Microsoft Visual Studio 2005 > Microsoft Visual Studio 2005**.
2. Open the **File** menu, point to **New**, and click **Project**.
3. Under **Project Types**, select **Visual C++**.
4. In the **Templates** section, select **Win32 Console Application**.
5. In the **Name** text box type the name of your project, for example, **Hello,World!**.
6. Choose a directory for your project. The default, **C:\Documents and Settings\Your Name\My Documents\Visual Studio 2005\Projects**, is usually a good choice.
7. Click **OK**.
8. The WIN32 Application Wizard should appear.
9. Select **Application Settings** on the left side of the dialog box.
10. Under **Additional Options** select **Empty Project**.
11. Click **Finish**. All compiler settings should now be initialized for your console project.

### C.3.2 Use the **std\_lib\_facilities.h** header file

For your first programs, we strongly suggest that you use the custom header file **std\_lib\_facilities.h** from [www.stroustrup.com/Programming/std\\_lib\\_facilities.h](http://www.stroustrup.com/Programming/std_lib_facilities.h).

Place a copy of it in the directory you chose in §C.3.1, step 6. (Note: Save as text, not HTML.) To use it, you need the line

```
#include "../std_lib_facilities.h"
```

in your program. The “*..../*” tells the compiler that you placed the header in **C:\Documents and Settings\Your Name\My Documents\Visual Studio 2005\Projects** where it can be used by all of your projects, rather than right next to your source file in a project where you would have to copy it for each project.

### C.3.3 Add a C++ source file to the project

You need at least one source file in your program (and often many):

1. Click the **Add New Item** icon on the menu bar (usually the second icon from the left). That will open the **Add New Item** dialog box. Select **Code** under the **Visual C++** category.
2. Select the **C++ File (.cpp)** icon in the template window. Type the name of your program file (**Hello,World!**) in the **Name** text box and click **Add**.

You have created an empty source code file. You are now ready to type your source code program.

### C.3.4 Enter your source code

At this point you can either enter the source code by typing it directly into the IDE, or you can copy and paste it from another source.

### C.3.5 Build an executable program

When you believe you have properly entered the source code for your program, go to the **Build** menu and select **Build Solution** or hit the triangular icon pointing to the right on the list of icons near the top of the IDE window. The IDE will try to compile and link your program. If it is successful, the message

**Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped**

should appear in the **Output** window. Otherwise a number of error messages will appear. Debug the program to correct the errors and **Build Solution** again.

If you used the triangular icon, the program will automatically start running (executing) if there were no errors. If you used the **Build Solution** menu item, you have to explicitly start the program, as described in §C.3.6.

### C.3.6 Execute the program

Once all errors have been eliminated, execute the program by going to the **Debug** menu and selecting **Start Without Debugging**.

### C.3.7 Save the program

Under the **File** menu, click **Save All**. If you forget and try to close the IDE, the IDE will remind you.

## C.4 Later

The IDE has an apparent infinity of features and options. Don’t worry about those early on – or you’ll get completely lost. If you manage to mess up a project so that it “behaves oddly,” ask an experienced friend for help or build a new project from scratch. Over time, slowly experiment with new features and options.





## Installing FLTK

“If the code and the comments disagree,  
then both are probably wrong.”

—Norm Schryer

This appendix describes how to download, install, and link to the FLTK graphics and GUI toolkit.

- D.1 Introduction**
- D.2 Downloading FLTK**
- D.3 Installing FLTK**
- D.4 Using FLTK in Visual Studio**
- D.5 Testing if it all worked**

## D.1 Introduction

We chose FLTK, the Fast Light Tool Kit (pronounced “full tick”), as the base for our presentation of graphics and GUI issues because it is portable, relatively simple, relatively conventional, and relatively easy to install. We explain how to install FLTK under Microsoft Visual Studio because that’s what most of our students use and because it is the hardest. If you use some other system (as some of our students also do), just look in the main folder (directory) of the downloaded files (§D.3) for directions for your favorite system.

Whenever you use a library that is not part of the ISO C++ standard, you (or someone else) have to download it, install it, and correctly use it from your own code. That’s rarely completely trivial, and installing FLTK is probably a good exercise – because downloading and installing even the best library can be quite frustrating when you haven’t tried before. Don’t be too reluctant to ask advice from people who have tried before, but don’t just let them do it for you: learn from them.

Note that there might be slight differences in files and procedures from what we describe here. For example, there may be a new version of FLTK or you may be using a different version of Visual Studio from what we describe in §D.4 or a completely different C++ implementation.

## D.2 Downloading FLTK

Before doing anything, first see if FLTK is already installed on your machine; see §D.5. If it is not there, the first thing to do is to get the files onto your computer:

1. Go to <http://fltk.org>. (In an emergency, instead download a copy from our book support website: [www.stroustrup.com/Programming/FLTK](http://www.stroustrup.com/Programming/FLTK).)
2. Click **Download** in the navigation menu.
3. Choose **FLTK 1.1.x** in the drop-down and click **Show Download Locations**.
4. Choose a download location and download the .zip file.

The file you get will be in .zip format. That is a compressed format suitable for transmitting lots of files across the net. You’ll need a program on your machine to “unzip” it into normal files; on Windows, WinZip and 7-Zip are examples of such programs.

## D.3 Installing FLTK

Your main problem in following our instructions is likely to be one of two: something has changed since we wrote and tested them (it happens), or the terminology is alien to you (we can't help with that; sorry). In the latter case, find a friend to translate.

1. Unzip the downloaded file and open the main folder, **fltk-1.1.?**. In a Visual C++ folder (e.g., **vc2005** or **vcnet**), open **fltk.dsw**. If asked about updating old project files, choose **Yes to All**.
2. From the **Build** menu, choose **Build Solution**. This may take a few minutes. The source code is being compiled into static link libraries so that you do not have to recompile the FLTK source code any time you make a new project. When the process has finished, close Visual Studio.
3. From the main FLTK directory open the **lib** folder. Copy (not just move/drag) all the **.lib** files except **README.lib** (there should be seven) into **C:\Program Files\Microsoft Visual Studio\Vc\lib**.
4. Go back to the FLTK main directory and copy the **FL** folder into **C:\Program Files\Microsoft Visual Studio\Vc\include**.

Experts will tell you that there are better ways to install than copying into **C:\Program Files\Microsoft Visual Studio\Vc\lib** and **C:\Program Files\Microsoft Visual Studio\Vc\include**. They are right, but we are not trying to make you VS experts. If the experts insist, let them be responsible for showing you the better alternative.

## D.4 Using FLTK in Visual Studio

1. Create a new project in Visual Studio with one change to the usual procedure: create a “Win32 project” instead of a “console application” when choosing your project type. Be sure to create an “empty project”; otherwise, some “software wizard” will add a lot of stuff to your project that you are unlikely to need or understand.
2. In Visual Studio, choose **Project** from the main (top) menu, and from the drop-down menu choose **Properties**.
3. In the **Properties** dialog box, in the left menu, click the **Linker** folder. This expands a sub-menu. In this sub-menu, click **Input**. In the **Additional Dependencies** text field on the right, enter the following text:

**fltkd.lib wsock32.lib comctl32.lib fltkjpegd.lib fltkimagesd.lib**

[The following step may be unnecessary because it is now the default.] In the **Ignore Specific Library** text field, enter the following text:

**libcd.lib**

4. [This step may be unnecessary because /MDd is now the default.] In the left menu of the same **Properties** window, click **C/C++** to expand a different sub-menu. Click the **Code Generation** sub-menu item. In the right menu, change the **Runtime Library** drop-down to **Multi-threaded Debug DLL (/MDd)**. Click OK to close the **Properties** window.

## D.5 Testing if it all worked

Create a single new .cpp file in your newly created project and enter the following code. It should compile without problems.

```
#include <FL/Fl.h>
#include <FL/Fl_Box.h>
#include <FL/Fl_Window.h>

int main()
{
    Fl_Window window(200, 200, "Window title");
    Fl_Box box(0,0,200,200, "Hey, I mean, Hello, World!");
    window.show();
    return Fl::run();
}
```

If it did not work:

- “Compiler error stating a .lib file could not be found”: Your problem is most likely in the installation section. Pay attention to step 3, which involves putting the link libraries (.lib) files where your compiler can easily find them.
- “Compiler error stating a .h file could not be opened”: Your problem is most likely in the installation section. Pay attention to step 4, which involves putting the header (.h) files where your compiler can easily find them.
- “Linker error involving unresolved external symbols”: Your problem is most likely in the project section.

If that didn’t help, find a friend to ask.



# GUI Implementation

“When you finally understand  
what you are doing,  
things will go right.”

—Bill Fairbank

This appendix presents implementation details of callbacks, **Window**, **Widget**, and **Vector\_ref**. In Chapter 16, we couldn’t assume the knowledge of pointers and casts needed for a more complete explanation, so we banished that explanation to this appendix.

- E.1 Callback implementation
- E.2 Widget implementation
- E.3 Window implementation
- E.4 Vector\_ref
- E.5 An example: manipulating Widgets

## E.1 Callback implementation

We implemented callbacks like this:

```
void Simple_window::cb_next(Address, Address addr)
    // call Simple_window::next() for the window located at addr
{
    reference_to<Simple_window>(addr).next();
}
```

Once you have understood Chapter 17, it is pretty obvious that an **Address** must be a **void\***. And, of course, **reference\_to<Simple\_window>(addr)** must somehow create a reference to a **Simple\_window** from the **void\*** called **addr**. However, unless you had previous programming experience, there was nothing “pretty obvious” or “of course” about that before you read Chapter 17, so let’s look at the use of addresses in detail.

As described in §A.17, C++ offers a way of giving a name to a type. For example:

```
typedef void* Address;           // Address is a synonym for void*
```

This means that the name **Address** can now be used instead of **void\***. Here, we used **Address** to emphasize that an address was passed, and also to hide the fact that **void\*** is the name of the type of pointer to an object for which we don’t know the type.

So **cb\_next()** receives a **void\*** called **addr** as an argument and – somehow – promptly converts it to a **Simple\_window&**:

```
reference_to<Simple_window>(addr)
```

The **reference\_to** is a template function (§A.13):

```
template<class W> W& reference_to(Address pw)
    // treat an address as a reference to a W
```

```

{
    return *static_cast<W*>(pw);
}

```

Here, we used a template function to write ourselves an operation that acts as a cast from a `void*` to a `Simple_window&`. The type conversion, `static_cast`, is described in §17.8.

The compiler has no way of verifying our assertion that `addr` points to a `Simple_window`, but the language rule requires the compiler to trust the programmer here. Fortunately, we are right. The way we know that we are right is that FLTK is handing us back a pointer that we gave to it. Since we knew the type of the pointer when we gave it to FLTK, we can use `reference_to` to “get it back.” This is messy, unchecked, and not all that uncommon at the lower levels of a system.

Once we have a reference to a `Simple_window`, we can use it to call a member function of `Simple_window`. For example (§16.3):

```

void Simple_window::cb_next(Address, Address pw)
    // call Simple_window::next() for the window located at pw
{
    reference_to<Simple_window>(pw).next();
}

```

We use the messy callback function `cb_next()` simply to adjust the types as needed to call a perfectly ordinary member function `next()`.

## E.2 Widget implementation

Our `Widget` interface class looks like this:

```

class Widget {
    // Widget is a handle to a Fl_widget — it is *not* a Fl_widget
    // we try to keep our interface classes at arm's length from FLTK
public:
    Widget(Point xy, int w, int h, const string& s, Callback cb)
        :loc(xy), width(w), height(h), label(s), do_it(cb)
    {}

    virtual ~Widget() {}           // destructor

    virtual void move(int dx,int dy)
        { hide(); pw->position(loc.x+=dx, loc.y+=dy); show(); }

```

```

virtual void hide() { pw->hide(); }
virtual void show() { pw->show(); }

virtual void attach(Window&) = 0; // each Widget defines at least
// one action for a window

Point loc;
int width;
int height;
string label;
Callback do_it;

protected:
Window* own; // every Widget belongs to a Window
FL_Widget* pw; // a Widget "knows" its FL_Widget
};

```

Note that our **Widget** keeps track of its FLTK widget and the **Window** with which it is associated. Note that we need pointers for that because a **Widget** can be associated with different **Windows** during its life. A reference or a named object wouldn't suffice. (Why not?)

It has a location (**loc**), a rectangular shape (**width** and **height**), and a **label**. Where it gets interesting is that it also has a callback function (**do\_it**) – it connects a **Widget**'s image on the screen to a piece of our code. The meaning of the operations (**move()**, **show()**, **hide()**, and **attach()**) should be obvious.

**Widget** has a “half-finished” look to it. It was designed as an implementation class that users should not have to see very often. It is a good candidate for a redesign. We are suspicious about all of those public data members, and “obvious” operations typically need to be reexamined for unplanned subtleties.

**Widget** has virtual functions and can be used as a base class, so it has a **virtual** destructor (§17.5.2).

### E.3 Window implementation

When do we use pointers and when do we use references instead? We examine that general question in §8.5.6. Here, we'll just observe that some programmers like pointers and that we need pointers when we want to point to different objects at different times in a program.

So far, we have not shown one of the central classes in our graphics and GUI library, **Window**. The most significant reasons are that it uses a pointer and that

its implementation using FLTK requires free store. As found in [Window.h](#), here it is:

```

class Window : public Fl_Window {
public:
    // let the system pick the location:
    Window(int w, int h, const string& title);
    // top left corner in xy:
    Window(Point xy, int w, int h, const string& title);

    virtual ~Window() {}

    int x_max() const { return w; }
    int y_max() const { return h; }

    void resize(int ww, int hh) { w=ww, h=hh; size(ww,hh); }

    void set_label(const string& s) { label(s.c_str()); }

    void attach(Shape& s) { shapes.push_back(&s); }
    void attach(Widget&);

    void detach(Shape& s);           // remove w from shapes
    void detach(Widget& w);         // remove w from window
                                    // (deactivates callbacks)

    void put_on_top(Shape& p);    // put p on top of other shapes
protected:
    void draw();
private:
    vector<Shape*> shapes;      // shapes attached to window
    int w,h;                      // window size

    void init();
};

```

So, when we **attach()** a **Shape** we store a pointer in **shapes** so that the **Window** can draw it. Since we can later **detach()** that shape, we need a pointer. Basically, an **attach()**ed shape is still owned by our code; we just give the **Window** a reference to it. **Window::attach()** converts its argument to a pointer so that it can store

it. As shown above, `attach()` is trivial; `detach()` is slightly less simple. Looking in `Window.cpp`, we find:

```
void Window::detach(Shape& s)
    // guess that the last attached will be first released
{
    for (vector<Shape*>::size_type i = shapes.size(); 0 < i; --i)
        if (shapes[i-1] == &s)
            shapes.erase(shapes.begin() + (i-1));
}
```

The `erase()` member function removes (“erases”) a value from a `vector`, decreasing the `vector`’s size by one (§20.7.1).

`Window` is meant to be used as a base class, so it has a `virtual` destructor (§17.5.2).

## E.4 Vector\_ref

Basically, `Vector_ref` simulates a `vector` of references. You can initialize it with references or with pointers:

- If an object is passed to `Vector_ref` as a reference, it is assumed to be owned by the caller who is responsible for its lifetime (e.g., the object is a scoped variable).
- If an object is passed to `Vector_ref` as a pointer, it is assumed to be allocated by `new` and it is `Vector_ref`’s responsibility to delete it.

An element is stored as a pointer – not as a copy of the object – into the `Vector_ref` and has reference semantics. For example, you can put a `Circle` into a `Vector_ref<Shape>` without suffering slicing.

```
template<class T> class Vector_ref {
    vector<T*> v;
    vector<T*> owned;
public:
    Vector_ref() {}
    Vector_ref(T* a, T* b = 0, T* c = 0, T* d = 0);

    ~Vector_ref() { for (int i=0; i < owned.size(); ++i) delete owned[i]; }

    void push_back(T& s) { v.push_back(&s); }
    void push_back(T* p) { v.push_back(p); owned.push_back(p); }
```

```

T& operator[](int i) { return *v[i]; }
const T& operator[](int i) const { return *v[i]; }

int size() const { return v.size(); }
};

```

**Vector\_ref**'s destructor **Deletes** every object passed to the **Vector\_ref** as a pointer.

## E.5 An example: manipulating Widgets

Here is a complete program. It exercises many of the **Widget/Window** features. It is only minimally commented. Unfortunately, such insufficient commenting is not uncommon. It is an exercise to get this program to run and to explain it.

Basically, when you run it, it appears to define four buttons:

```

#include "../GUI.h"
using namespace Graph_lib;

class W7 : public Window {
    // four ways to make it appear that a button moves around:
    // show/hide, change location, create new one, and attach/detach
public:
    W7(int w, int h, const string& t);

    Button* p1;           // show/hide
    Button* p2;
    bool sh_left;

    Button* mvp;          // move
    bool mv_left;

    Button* cdp;          // create/destroy
    bool cd_left;

    Button* adp1;         // activate/deactivate
    Button* adp2;
    bool ad_left;

    void sh();             // actions
    void mv();
    void cd();
    void ad();

```

```

static void cb_sh(Address, Address addr)           // callbacks
    { reference_to<W7>(addr).sh(); }
static void cb_mv(Address, Address addr)
    { reference_to<W7>(addr).mv(); }
static void cb_cd(Address, Address addr)
    { reference_to<W7>(addr).cd(); }
static void cb_ad(Address, Address addr)
    { reference_to<W7>(addr).ad(); }
};


```

However, a **W7** (Window experiment number 7) really has six buttons; it just keeps two hidden:

```

W7::W7(int w, int h, const string& t)
    :Window{w,h,t},
    sh_left{true}, mv_left{true}, cd_left{true}, ad_left{true}
{
    p1 = new Button{Point{100,100},50,20,"show",cb_sh};
    p2 = new Button{Point{200,100},50,20, "hide",cb_sh};

    mvp = new Button{Point{100,200},50,20,"move",cb_mv};

    cdp = new Button{Point{100,300},50,20,"create",cb_cd};

    adp1 = new Button{Point{100,400},50,20,"activate",cb_ad};
    adp2 = new Button{Point{200,400},80,20,"deactivate",cb_ad};

    attach(*p1);
    attach(*p2);
    attach(*mvp);
    attach(*cdp);
    p2->hide();
    attach(*adp1);
}

```

There are four callbacks. Each makes it appear that the button you press disappears and a new one appears. However, this is achieved in four different ways:

```

void W7::sh()           // hide a button, show another
{
    if (sh_left) {
        p1->hide();
        p2->show();
    }
}

```

```
    else {
        p1->show();
        p2->hide();
    }
    sh_left = !sh_left;
}

void W7::mv() // move the button
{
    if (mv_left) {
        mvp->move(100,0);
    }
    else {
        mvp->move(-100,0);
    }
    mv_left = !mv_left;
}

void W7::cd() // delete the button and create a new one
{
    cdp->hide();
    delete cdp;
    string lab = "create";
    int x = 100;
    if (cd_left) {
        lab = "delete";
        x = 200;
    }
    cdp = new Button{Point{x,300}, 50, 20, lab, cb_cd};
    attach(*cdp);
    cd_left = !cd_left;
}

void W7::ad() // detach the button from the window and attach its replacement
{
    if (ad_left) {
        detach(*adp1);
        attach(*adp2);
    }
    else {
        detach(*adp2);
        attach(*adp1);
    }
}
```

```
    ad_left = !ad_left;  
}  
  
int main()  
{  
    W7 w{400,500,"move"};  
    return gui_main();  
}
```

This program demonstrates the fundamental ways of adding and subtracting widgets to/from a window – or just appearing to.

# Glossary

“Often, a few well-chosen words  
are worth a thousand pictures.”

—Anonymous

A *glossary* is a brief explanation of words used in a text. This is a rather short glossary of the terms we thought most essential, especially in the earlier stages of learning programming. The index and the “Terms” sections of the chapters might also help. A more extensive glossary, relating specifically to C++, can be found at [www.stroustrup.com/glossary.html](http://www.stroustrup.com/glossary.html), and there is an incredible variety of specialized glossaries (of greatly varying quality) available on the web. Please note that a term can have several related meanings (so we occasionally list some) and that most terms we list have (often weakly) related meanings in other contexts; for example, we don’t define *abstract* as it relates to modern painting, legal practice, or philosophy.

**abstract class** a class that cannot be directly used to create objects; often used to define an interface to derived classes. A class is made abstract by having a pure virtual function or a protected constructor.

**abstraction** a description of something that selectively and deliberately ignores (hides) details (e.g., implementation details); selective ignorance.

**address** a value that allows us to find an object in a computer’s memory.

**algorithm** a procedure or formula for solving a problem; a finite series of computational steps to produce a result.

**alias** an alternative way of referring to an object; often a name, pointer, or reference.

**application** a program or a collection of programs that is considered an entity by its users.

**approximation** something (e.g., a value or a design) that is close to the perfect or ideal (value or design). Often an approximation is a result of trade-offs among ideals.

**argument** a value passed to a function or a template, in which it is accessed through a parameter.

**array** a homogeneous sequence of elements, usually numbered, e.g., [0:max).

**assertion** a statement inserted into a program to state (assert) that something must always be true at this point in the program.

**base class** a class used as the base of a class hierarchy. Typically a base class has one or more virtual functions.

**bit** the basic unit of information in a computer. A bit can have the value 0 or the value 1.

**bug** an error in a program.

**byte** the basic unit of addressing in most computers. Typically, a byte holds 8 bits.

**class** a user-defined type that may contain data members, function members, and member types.

**code** a program or a part of a program; ambiguously used for both source code and object code.

**compiler** a program that turns source code into object code.

**complexity** a hard-to-precisely-define notion or measure of the difficulty of constructing a solution to a problem or of the solution itself. Sometimes *complexity* is used to (simply) mean an estimate of the number of operations needed to execute an algorithm.

**computation** the execution of some code, usually taking some input and producing some output.

**concept** (1) a notion, an idea; (2) a set of requirements, usually for a template argument.

**concrete class** a class for which objects can be created.

**constant** a value that cannot be changed (in a given scope); not mutable.

**constructor** an operation that initializes (“constructs”) an object. Typically a constructor establishes an invariant and often acquires resources needed for an object to be used (which are then typically released by a destructor).

**container** an object that holds elements (other objects).

**copy** an operation that makes two objects have values that compare equal. See also **move**.

**correctness** a program or a piece of a program is correct if it meets its specification. Unfortunately, a specification can be incomplete or inconsistent, or can fail to meet users’ reasonable expectations. Thus, to produce acceptable code, we sometimes have to do more than just follow the formal specification.

**cost** the expense (e.g., in programmer time, run time, or space) of producing a program or of executing it. Ideally, cost should be a function of complexity.

**data** values used in a computation.

**debugging** the act of searching for and removing errors from a program; usually far less systematic than testing.

**declaration** the specification of a name with its type in a program.

**definition** a declaration of an entity that supplies all information necessary to complete a program using the entity. Simplified definition: a declaration that allocates memory.

**derived class** a class derived from one or more base classes.

**design** an overall description of how a piece of software should operate to meet its specification.

**destructor** an operation that is implicitly invoked (called) when an object is destroyed (e.g., at the end of a scope). Often, it releases resources.

**encapsulation** protecting something meant to be private (e.g., implementation details) from unauthorized access.

**error** a mismatch between reasonable expectations of program behavior (often expressed as a requirement or a users' guide) and what a program actually does.

**executable** a program ready to be run (executed) on a computer.

**feature creep** a tendency to add excess functionality to a program "just in case."

**file** a container of permanent information in a computer.

**floating-point number** a computer's approximation of a real number, such as 7.93 and 10.78e-3.

**function** a named unit of code that can be invoked (called) from different parts of a program; a logical unit of computation.

**generic programming** a style of programming focused on the design and efficient implementation of algorithms. A generic algorithm will work for all argument types that meet its requirements. In C++, generic programming typically uses templates.

**handle** a class that allows access to another through a member pointer or reference. See also **copy**, **move**, **resource**.

**header** a file containing declarations used to share interfaces between parts of a program.

**hiding** the act of preventing a piece of information from being directly seen or accessed. For example, a name from a nested (inner) scope can prevent that same name from an outer (enclosing) scope from being directly used.

**ideal** the perfect version of something we are striving for. Usually we have to make trade-offs and settle for an approximation.

**implementation** (1) the act of writing and testing code; (2) the code that implements a program.

**infinite loop** a loop where the termination condition never becomes true. See **iteration**.

- infinite recursion** a recursion that doesn't end until the machine runs out of memory to hold the calls. In reality, such recursion is never infinite but is terminated by some hardware error.
- information hiding** the act of separating interface and implementation, thus hiding implementation details not meant for the user's attention and providing an abstraction.
- initialize** giving an object its first (initial) value.
- input** values used by a computation (e.g., function arguments and characters typed on a keyboard).
- integer** a whole number, such as 42 and -99.
- interface** a declaration or a set of declarations specifying how a piece of code (such as a function or a class) can be called.
- invariant** something that must be always true at a given point (or points) of a program; typically used to describe the state (set of values) of an object or the state of a loop before entry into the repeated statement.
- iteration** the act of repeatedly executing a piece of code; see **recursion**.
- iterator** an object that identifies an element of a sequence.
- library** a collection of types, functions, classes, etc. implementing a set of facilities (abstractions) meant to be potentially used as part of more than one program.
- lifetime** the time from the initialization of an object until it becomes unusable (goes out of scope, is deleted, or the program terminates).
- linker** a program that combines object code files and libraries into an executable program.
- literal** a notation that directly specifies a value, such as 12 specifying the integer value "twelve."
- loop** a piece of code executed repeatedly; in C++, typically a **for**-statement or a **while**-statement.
- move** an operation that transfers a value from one object to another, leaving behind a value representing "empty." See also **copy**.
- mutable** changeable; the opposite of immutable, constant, and variable.
- object** (1) an initialized region of memory of a known type which holds a value of that type; (2) a region of memory.
- object code** output from a compiler intended as input for a linker (for the linker to produce executable code).
- object file** a file containing object code.
- object-oriented programming** a style of programming focused on the design and use of classes and class hierarchies.
- operation** something that can perform some action, such as a function and an operator.
- output** values produced by a computation (e.g., a function result or lines of characters written on a screen).
- overflow** producing a value that cannot be stored in its intended target.

- overload** defining two functions or operators with the same name but different argument (operand) types.
- override** defining a function in a derived class with the same name and argument types as a virtual function in the base class, thus making the function callable through the interface defined by the base class.
- owner** an object responsible for releasing a resource.
- paradigm** a somewhat pretentious term for design or programming style; often used with the (erroneous) implication that there exists a paradigm that is superior to all others.
- parameter** a declaration of an explicit input to a function or a template. When called, a function can access the arguments passed through the names of its parameters.
- pointer** (1) a value used to identify a typed object in memory; (2) a variable holding such a value.
- post-condition** a condition that must hold upon exit from a piece of code, such as a function or a loop.
- pre-condition** a condition that must hold upon entry into a piece of code, such as a function or a loop.
- program** code (possibly with associated data) that is sufficiently complete to be executed by a computer.
- programming** the art of expressing solutions to problems as code.
- programming language** a language for expressing programs.
- pseudo code** a description of a computation written in an informal notation rather than a programming language.
- pure virtual function** a virtual function that must be overridden in a derived class.
- RAII (“Resource Acquisition Is Initialization”)** a basic technique for resource management based on scopes.
- range** a sequence of values that can be described by a start point and an end point. For example, [0:5) means the values 0, 1, 2, 3, and 4.
- recursion** the act of a function calling itself; see also **iteration**.
- reference** (1) a value describing the location of a typed value in memory; (2) a variable holding such a value.
- regular expression** a notation for patterns in character strings.
- requirement** (1) a description of the desired behavior of a program or part of a program; (2) a description of the assumptions a function or template makes of its arguments.
- resource** something that is acquired and must later be released, such as a file handle, a lock, or memory. See also **handle**, **owner**.
- rounding** conversion of a value to the mathematically nearest value of a less precise type.
- scope** the region of program text (source code) in which a name can be referred to.
- sequence** elements that can be visited in a linear order.

- software** a collection of pieces of code and associated data; often used interchangeably with **program**.
- source code** code as produced by a programmer and (in principle) readable by other programmers.
- source file** a file containing source code.
- specification** a description of what a piece of code should do.
- standard** an officially agreed-upon definition of something, such as a programming language.
- state** a set of values.
- string** a sequence of characters.
- style** a set of techniques for programming leading to a consistent use of language features; sometimes used in a very restricted sense to refer just to low-level rules for naming and appearance of code.
- subtype** derived type; a type that has all the properties of a type and possibly more.
- supertype** base type; a type that has a subset of the properties of a type.
- system** (1) a program or a set of programs for performing a task on a computer; (2) a shorthand for “operating system,” that is, the fundamental execution environment and tools for a computer.
- template** a class or a function parameterized by one or more types or (compile-time) values; the basic C++ language construct supporting generic programming.
- testing** a systematic search for errors in a program.
- trade-off** the result of balancing several design and implementation criteria.
- truncation** loss of information in a conversion from a type into another that cannot exactly represent the value to be converted.
- type** something that defines a set of possible values and a set of operations for an object.
- uninitialized** the (undefined) state of an object before it is initialized.
- unit** (1) a standard measure that gives meaning to a value (e.g., km for a distance); (2) a distinguished (e.g., named) part of a larger whole.
- use case** a specific (typically simple) use of a program meant to test its functionality and demonstrate its purpose.
- value** a set of bits in memory interpreted according to a type.
- variable** a named object of a given type; contains a value unless uninitialized.
- virtual function** a member function that can be overridden in a derived class.
- word** a basic unit of memory in a computer, usually the unit used to hold an integer.

# Bibliography

- Aho, Alfred V., Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools, Second Edition* (usually called “The Dragon Book”). Addison-Wesley, 2006. ISBN 0321486811.
- Andrews, Mike, and James A. Whittaker. *How to Break Software: Functional and Security Testing of Web Applications and Web Services*. Addison-Wesley, 2006. ISBN 0321369440.
- Bergin, Thomas J., and Richard G. Gibson, eds. *History of Programming Languages, Volume 2*. Addison-Wesley, 1996. ISBN 0201895021.
- Blanchette, Jasmin, and Mark Summerfield. *C++ GUI Programming with Qt 4*. Prentice Hall, 2006. ISBN 0131872494.
- Boost.org. “A Repository for Libraries Meant to Work Well with the C++ Standard Library.” [www.boost.org](http://www.boost.org).
- Cox, Russ. “Regular Expression Matching Can Be Simple and Fast (but Is Slow in Java, Perl, PHP, Python, Ruby, . . .).” <http://swtch.com/~rsc/regexp/regexp1.html>.
- dmoz.org. <http://dmoz.org/Computers/Programming/Languages>.
- Freeman, T. L., and C. Phillips. *Parallel Numerical Algorithms*. Prentice Hall, 1992. ISBN 0136515975.
- Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994. ISBN 0201633612.
- Goldthwaite, Lois, ed. *Technical Report on C++ Performance*. ISO/IEC PDTR 18015. [www.stroustrup.com/performanceTR.pdf](http://www.stroustrup.com/performanceTR.pdf).
- Gullberg, Jan. *Mathematics – From the Birth of Numbers*. W. W. Norton, 1996. ISBN 039304002X.
- Hailpern, Brent, and Barbara G. Ryder, eds. *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- ISO/IEC 9899:2011. *Programming Languages – C*. The C standard.
- ISO/IEC 14882:2011. *Programming Languages – C++*. The C++ standard.
- Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language, Second Edition*. Prentice Hall, 1988. ISBN 0131103628.

- Knuth, Donald E. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms, Third Edition*. Addison-Wesley, 1997. ISBN 0201896842.
- Koenig, Andrew, and Barbara E. Moo. *Accelerated C++: Practical Programming by Example*. Addison-Wesley, 2000. ISBN 020170353X.
- Langer, Angelika, and Klaus Kreft. *Standard C++ IOStreams and Locales: Advanced Programmer's Guide and Reference*. Addison-Wesley, 2000. ISBN 0321585585.
- Lippman, Stanley B., José Lajoie, and Barbara E. Moo. *The C++ Primer, Fifth Edition*. Addison-Wesley, 2005. ISBN 0321714113. (Use only the 5th edition.)
- Lockheed Martin Corporation. "Joint Strike Fighter Air Vehicle Coding Standards for the System Development and Demonstration Program." Document Number 2RDU00001 Rev C. December 2005. Colloquially known as "JSF++." [www.stroustrup.com/JSF-AV-rules.pdf](http://www.stroustrup.com/JSF-AV-rules.pdf).
- Lohr, Steve. *Go To: The Story of the Math Majors, Bridge Players, Engineers, Chess Wizards, Maverick Scientists and Iconoclasts – The Programmers Who Created the Software Revolution*. Basic Books, 2002. ISBN 978-0465042265.
- Meyers, Scott. *Effective STL: 50 Specific Ways to Improve Your Use of the Standard Template Library*. Addison-Wesley, 2001. ISBN 0201749629.
- Meyers, Scott. *Effective C++: 55 Specific Ways to Improve Your Programs and Designs, Third Edition*. Addison-Wesley, 2005. ISBN 0321334876.
- Programming Research. *High-Integrity C++ Coding Standard Manual Version 2.4*. [www.programmingresearch.com](http://www.programmingresearch.com).
- Richards, Martin. *BCPL – The Language and Its Compiler*. Cambridge University Press, 1980. ISBN 0521219655.
- Ritchie, Dennis. "The Development of the C Programming Language." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. ACM SIGPLAN Notices, Vol. 28 No. 3, 1993.
- Salus, Peter H. *A Quarter Century of UNIX*. Addison-Wesley, 1994. ISBN 0201547775.
- Sammet, Jean E. *Programming Languages: History and Fundamentals*. Prentice Hall, 1969. ISBN 0137299885.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*. Addison-Wesley, 2002. ISBN 0201604647.
- Schmidt, Douglas C., and Stephen D. Huston. *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*. Addison-Wesley, 2003. ISBN 0201795256.
- Schwartz, Randal L., Tom Phoenix, and Brian D. Foy. *Learning Perl, Fourth Edition*. O'Reilly, 2005. ISBN 0596101058.
- Scott, Michael L. *Programming Language Pragmatics*. Morgan Kaufmann, 2000. ISBN 1558604421.
- Sebesta, Robert W. *Concepts of Programming Languages, Sixth Edition*. Addison-Wesley, 2003. ISBN 0321193628.
- Shepherd, Simon. "The Tiny Encryption Algorithm (TEA)." [www.tayloredge.com/reference/Mathematics/TEA-XTEA.pdf](http://www.tayloredge.com/reference/Mathematics/TEA-XTEA.pdf) and <http://143.53.36.235:8080/tea.htm>.
- Stepanov, Alexander. [www.stepanovpapers.com](http://www.stepanovpapers.com).
- Stewart, G. W. *Matrix Algorithms, Volume I: Basic Decompositions*. SIAM, 1998. ISBN 0898714141.
- Stone, Debbie, Caroline Jarrett, Mark Woodroffe, and Shailey Minocha. *User Interface Design and Evaluation*. Morgan Kaufmann, 2005. ISBN 0120884364.

- Stroustrup, Bjarne. "A History of C++: 1979–1991." *Proceedings of the ACM History of Programming Languages Conference (HOPL-2)*. ACM SIGPLAN Notices, Vol. 28 No. 3, 1993.
- Stroustrup, Bjarne. *The Design and Evolution of C++*. Addison-Wesley, 1994. ISBN 0201543303.
- Stroustrup, Bjarne. "Learning Standard C++ as a New Language." *C/C++ Users Journal*, May 1999.
- Stroustrup, Bjarne. "C and C++: Siblings"; "C and C++: A Case for Compatibility"; and "C and C++: Case Studies in Compatibility." *The C/C++ Users Journal*, July, Aug., and Sept. 2002.
- Stroustrup, Bjarne. "Evolving a Language in and for the Real World: C++ 1991–2006." *Proceedings of the Third ACM SIGPLAN Conference on the History of Programming Languages (HOPL-III)*. San Diego, CA, 2007. <http://portal.acm.org/toc.cfm?id=1238844>.
- Stroustrup, Bjarne. *The C++ Programming Language, Fourth Edition*. Addison-Wesley, 2013. ISBN 0321563840.
- Stroustrup, Bjarne. *A Tour of C++*. Addison-Wesley, 2013. ISBN 978-0321958310.
- Stroustrup, Bjarne. Author's home page, [www.stroustrup.com](http://www.stroustrup.com).
- Sutter, Herb. *Exceptional C++: 47 Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley, 2000. ISBN 0201615622.
- Sutter, Herb, and Andrei Alexandrescu. *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley, 2004. ISBN 0321113586.
- University of St. Andrews. The MacTutor History of Mathematics archive. <http://www-gap.dcs.st-and.ac.uk/~history>.
- Wexelblat, Richard L., ed. *History of Programming Languages*. Academic Press, 1981. ISBN 0127450408.
- Whittaker, James A. *How to Break Software: A Practical Guide to Testing*. Addison-Wesley, 2002. ISBN 0201796198.
- Wood, Alastair. *Introduction to Numerical Analysis*. Addison-Wesley, 2000. ISBN 020134291X.



# Index

- !. See Not*, 1087
- !=. See Not equal (inequality)*, 67, 1088, 1101
- "...". See String literal*, 62
- #. See Preprocessor directives*, 1129
- \$ . See End of line*, 873, 1178
- % . See*
  - Output format specifier, 1187
  - Remainder (modulo), 68
- %=. See Remainder and assign*, 1090
- & . See*
  - Address of, 588, 1087
  - Bitwise logical operations (and), 956, 1089, 1094
  - Reference to (in declarations), 276–279, 1099
- &&. See Logical and*, 1089, 1094
- &=. See Bitwise logical operations (and and assign)*, 1090
- .!. . See Character literals*, 161, 1079–1080
- (. See*
  - Expression (grouping), 95, 867, 873, 876
  - Function call, 285, 766
  - Function of (in declarations), 113–115, 1099
  - Regular expression (grouping), 1178
- \*. See*
  - Contents of (dereference), 594
  - Multiply, 1088
  - Pointer to (in declarations), 587, 1099
  - Repetition (in **regex**), 868, 873–874, 1178
- \*/ end of block comment*, 238
- \*=. See Multiply and assign (scale)*, 67
- + . See*
  - Add, 66, 1088
  - Concatenation (of **strings**), 68–69, 851, 1176
  - Repetition in **regex**, 873–875, 1178
- ++. See Increment*, 66, 721
- +=. See*
  - Add and assign, 1089
  - Move forward, 1101
- string** (add at end)*, 851, 1176
- , (comma). See*
  - Comma operator, 1090
  - List separator, 1103, 1122–1123
- . See*
  - Minus (subtraction), 66, 1088
  - Regular expression (range), 877
- . See Decrement*, 66, 1087, 1141
- > (arrow). See Member access*, 608, 1087, 1109, 1141
- = See*
  - Move backward, 1101, 1142
  - Subtract and assign, 67, 1090
- . (dot). See*
  - Member access, 306, 607–608, 1086–1087
  - Regular expression, 872, 1178
- ... (ellipsis). See*
  - Arguments (unchecked), 1105–1106
  - Catch all exceptions, 152
- . See Divide, 66, 1088
- // . See Line comment*, 45
- /\* . . \*/. See Block comment*, 238
- /=. See Divide and assign*, 67, 1090
- : (colon). See*
  - Base and member initializers, 315, 477, 555
  - Conditional expression, 268
  - Label, 106–108, 306, 511, 1096
- ::. See Scope (resolution)*, 295, 314, 1083, 1086
- ; (semicolon). See Statement (terminator)*, 50, 100

- <. See** Less than, 67, 1088
- <<. See**
  - Bitwise logical operations (left shift), 956, 1088
  - Output, 363–365, 1173
- <<. See** Less than or equal, 67, 1088
- <<<. See** Bitwise logical operations (shift left and assign), 1090
- < . >. See** Template (arguments and parameters), 153, 678–679
- =. See**
  - Assignment, 66, 1090
  - Initialization, 69–73, 1219
- ==. See** Equal, 67, 1088
- >. See**
  - Greater than, 67, 1088
  - Input prompt, 223
  - Template (argument-list terminator), 679
- >>. See** Greater than or equal, 67, 1088
- >>. See**
  - Bitwise logical operations (right shift), 956, 1088
  - Input, 61, 365
- >>>. See** Bitwise logical operations (shift right and assign), 1090
- ?.** See
  - Conditional expression, 268, 1089
  - Regular expression, 867–868, 873, 874–875, 1178
- [].** See
  - Array of (in declaration), 649, 1099
  - Regular expression (character class), 872, 1178
  - Subscripting, 594, 649, 1101
- \(backslash).** See
  - Character literal, 1079–1080
  - Escape character, 1178
  - Regular expression (escape character), 866–867, 873, 877
- ^.** See
  - Bitwise logical operations (exclusive or), 956, 1089, 1094
  - Regular expression (not), 873, 1178
- ^=.** See Bitwise logical operations (xor and assign), 1090
- \_.** See Underscore, 75, 76, 1081
- {.** See
  - Block delimiter, 47, 111
  - Initialization, 83
  - List, 83
  - Regular expression (range), 867, 873–875, 1178
- |.** See
  - Bitwise logical operations (bitwise or), 956, 1089, 1094
  - Regular expression (or), 867–868, 873, 876, 1178
- |=. See** Bitwise logical operations (or and assign), 1090
- ||. See** Logical or, 1089, 1094
- ~.** See
  - Bitwise logical operations (complement), 956, 1087
  - Destructors, 601–603
- 0** (zero). *See*
  - Null pointer, 598
  - Prefix, 382, 384
  - printf()** format specifier, 1188–1189
- 0x.** *See* Prefix, 382, 384

## A

- a**, append file mode, 1186
- \a** alert, character literal, 1079
- abort()**, 1194–1195
- abs()**, absolute value, 917, 1181
- complex**, 920, 1183
- Abstract classes, 495, 1217
  - class hierarchies, 512
  - creating, 495, 512, 1118–1119
- Shape** example, 495–496
- Abstract-first approach to programming, 10
- Abstraction, 92–93, 1217
  - level, ideals, 812–813
- Access control, 306, 505, 511
  - base classes, 511
  - encapsulation, 505
  - members, 492–493
  - private, 505, 511
  - private by default, 306–307
  - private:** label, 306
  - private *vs.* public, 306–308
  - protected, 505, 511
  - protected:** label, 511
  - public, 306, 505, 511
  - public by default, 307–308. *See also struct*
  - public:** label, 306
- Shape** example, 496–499
- accumulate()**, 759, 770–772, 1183
  - accumulator, 770
  - generalizing, 772–774
- acos()**, arccosine, 917, 1182

Action, 47  
Activation record, 287. *See also* Stacks  
Ada language, 832–833  
Adaptors  
    **bind()**, 1164  
    container, 1144  
    function objects, 1164  
    **mem\_fn()**, 1164  
    **not1()**, 1164  
    **not2()**, 1164  
    **priority\_queue**, 1144  
    **queue**, 1144  
    **stack**, 1144  
    **add()**, 449–450, 491–492, 615–617  
Add (plus) **+**, 66, 1088  
Add and assign **+=**, 66, 73, 1090  
Additive operators, 1088  
Address, 588, 1217  
    unchecked conversions, 943–944  
Address of (unary) **&**, 588, 1087  
Ad hoc polymorphism, 682–683  
    **adjacent\_difference()**, 770, 1184  
    **adjacent\_find()**, 1153  
    **advance()**, 615–617, 739, 1142  
Affordability, software, 34  
Age distribution example, 538–539  
Alert markers, 3  
Algol60 language, 827–829  
Algol family of languages, 826–829  
    **<algorithm>**, 759, 1133  
Algorithms, 1217  
    and containers, 722  
    header files, 1133–1134  
    numerical, 1183–1184  
    passing arguments to. *See* Function objects  
Algorithms, numerical, 770, 1183–1184  
    **accumulate()**, 759, 770–774, 1183  
    **adjacent\_difference()**, 770, 1184  
    **inner\_product()**, 759, 770, 774–776, 1184  
    **partial\_sum()**, 770, 1184  
Algorithms, STL, 1152–1153  
    **<algorithm>**, 759  
    **binary\_search()**, 796  
    comparing elements, 759  
    **copy()**, 758, 789–790  
    **copy\_if()**, 789  
    copying elements, 758  
    **count()**, 758  
    **count\_if()**, 758  
    **equal()**, 759  
    **equal\_range()**, 758, 796  
    **find()**, 758, 759–763  
    **find\_if()**, 758, 763–764  
    heap, 1160  
    **lower\_bound()**, 796  
    **max()**, 1161  
    **merge()**, 758  
    merging sorted sequences, 758  
    **min()**, 1161  
    modifying sequence, 1154–1156  
    mutating sequence, 1154–1156  
    nonmodifying sequence, 1153–1154  
    numerical. *See* Algorithms, numerical  
    permutations, 1160–1161  
    **search()**, 795–796  
    searching, 1157–1159. *See also* **find\_if()**; **find()**  
    set, 1159–1160  
    **shuffle()**, 1155–1156  
    **sort()**, 758, 794–796  
    sorting, 758, 794–796, 1157–1159  
    summing elements, 759  
    testing, 1001–1008  
    **unique\_copy()**, 758, 789, 792–793  
    **upper\_bound()**, 796  
    utility, 1157  
    value comparisons, 1161–1162  
Aliases, 1128, 1217. *See also* References  
Allocating memory. *See also* Deallocating memory;  
    Memory  
    **allocator\_type**, 1147  
    **bad\_alloc** exception, 1094  
C++ and C, 1043–1044  
    **calloc()**, 1193  
    embedded systems, 935–936, 940–942  
    free store, 593–594  
    **malloc()**, 1043–1044, 1193  
    **new**, 1094–1095  
    pools, 940–941  
    **realloc()**, 1045  
    stacks, 942–943  
    **allocator\_type**, 1147  
Almost containers, 751, 1145  
    **alnum**, **regex** character class, 878, 1179  
    **alpha**, **regex** character class, 878, 1179  
Alternation  
    patterns, 194  
    regular expressions, 876  
Ambiguous function call, 1104  
Analysis, 35, 176, 179  
**and**, synonym for **&**, 1037, 1038

**and\_eq**, synonym for `&=`, 1037, 1038  
**app** mode, 389, 1170  
**append()**, 851, 1177  
Append  
  files, 389, 1186  
  string `+=`, 851  
Application  
  collection of programs, 1218  
  operator `( )`, 766  
Approximation, 532–537, 1218  
Arccosine, **acos()**, 917  
Arcsine, **asin()**, 918  
Arctangent, **atan()**, 918  
**arg()**, of complex number, theta, 920, 1183  
Argument deduction, 689–690  
Argument errors  
  callee responsibility, 143–145  
  caller responsibility, 142–143  
  reasons for, 144–145  
Arguments, 272, 1218  
  formal. *See* Parameters  
  functions, 1105–1106  
  passing. *See* Passing arguments  
  program input, 91  
  source of exceptions, 147–148  
  templates, 1122–1123  
  types, class interfaces, 324–326  
  unchecked, 1029–1030, 1105–1106  
  unexpected, 136  
Arithmetic if `?:`, 268. *See also* Conditional expression `?:`  
Arithmetic operations. *See* Numerics  
**<array>**, 1133  
Arrays, 648–650, 1218. *See also* Containers; **vector**  
  **declaration**, 649  
  **dereferencing**, 649  
  accessing elements, 649, 899–901  
  assignment, 653–654  
  associative. *See* Associative containers  
  built-in, 747–749  
  copying, 653–654  
  C-style strings, 654–655  
  dereferencing, 649  
  element numbering, 649  
  initializing, 596–598, 654–656  
  multidimensional, 895–897, 1102  
  palindrome example, 660–661  
  passing pointers to arrays, 944–951  
  pointers to elements, 650–652  
range checking, 649  
subscripting `[]`, 649  
terminating zero, 654–655  
**vector** alternative, 947–951  
Arrays and pointers, 651–658  
  debugging, 656–659  
**array** standard library class, 747–749, 1144  
**asin()**, arcsine, 918, 1182  
**asm()**, assembler insert, 1037  
Assemblers, 820  
Assertions  
  **assert()**, 1061  
  **<cassert>**, 1135  
  debugging, 163  
  definition, 1218  
**assign()**, 1148  
Assignment `=`, 69–73  
  arrays, 653–654  
  assignment and initialization, 69–73  
  composite assignment operators, 73–74  
  containers, 1148  
**Date** example, 309–310  
enumerators, 318–319  
expressions, 1089–1090  
**string**, 851  
**vector**, resizing, 675–677  
Assignment operators (composite), 66  
  `%=`, 73, 1090  
  `&=`, 1090  
  `*=`, 73, 1089  
  `+=`, 73, 1090, 1141  
  `-=`, 73, 1090, 1142  
  `/=`, 73, 1090  
  `<<=`, 1090  
  `>>=`, 1090  
  `^=`, 1090  
  `|=`, 1090  
Associative arrays. *See* Associative containers  
Associative containers, 776, 1144  
  email example, 856–860  
  header files, 776  
  map, 776  
  **multimap**, 776, 860–861  
  **multiset**, 776  
  operations, 1151–1152  
  **set**, 776  
  **unordered\_map**, 776  
  **unordered\_multimap**, 776  
  **unordered\_multiset**, 776  
  **unordered\_set**, 776

Assumptions, testing, 1009–1011  
**at()**, range-checked subscripting, 693–694, 1149  
**atan()**, arctangent, 918, 1182  
**ate** mode, 389, 1170  
**atof()**, string to **double**, 1192  
**atoi()**, string to **int**, 1192  
**atol()**, string to **long**, 1192  
AT&T Bell Labs, 838  
AT&T Labs, 838  
**attach()** *vs.* **add()** example, 491–492  
**auto**, 732–734, 760  
Automatic storage, 591–592, 1083. *See also*  
  Stack storage  
**Axis** example, 424–426, 443, 529–532, 543–546

## B

**b**, binary file mode, 1186  
Babbage, Charles, 832  
**back()**, last element, 737, 1149  
**back\_inserter()**, 1162  
Backus, John, 823  
Backus-Naur (BNF) Form, 823, 828  
**bad\_alloc** exception, 1094  
**bad()** stream state, 355, 1171  
Base-2 number system (binary), 1078–1079  
Base-8 number system (octal), 1077–1078  
Base-10  
  logarithms, 918  
  number system (decimal), 1077–1078  
Base-16 number system (hexadecimal), 1077–1078  
Balanced trees, 780–782  
Base and member initializers, 315, 477, 555  
Base classes, 493–496, 504–507, 1218  
  abstract classes, 495, 512–513, 1118–1119  
  access control, 511  
  derived classes, 1116–1117  
  description, 504–506  
  initialization of, 477, 555, 1113, 1117  
  interface, 513–514  
  object layout, 506–507  
  overriding, 508–511  
**Shape** example, 495–496  
virtual function calls, 501, 506–507  
**vptr**, 506  
**vtbl**, 506  
Base-e exponentials, 918

**basic\_string**, 852  
Basic guarantee, 702  
BCPL language, 838  
**begin()**  
  iterator, 1148  
**string**, 851, 1177  
**vector**, 721  
Bell Telephone Laboratories (Bell Labs), 836, 838–842, 1022–1023  
Bentley, John, 933, 966  
Bidirectional iterator, 1142  
**bidirectional** iterators, 752  
Big-O notation, complexity, 785  
Binary I/O, 390–393  
**binary** mode, 389, 1170  
Binary number system, 1078–1079  
Binary search, 758, 779, 795–796  
**binary\_search()**, 796, 1158  
**bind()** adaptor, 1164  
**bitand**, synonym for **&**, 1037, 1038  
Bitfields, 956–957, 967–969, 1120–1121  
**bitor**, synonym for **|**, 1038  
Bits, 78, 954, 1218  
  bitfields, 956–957  
**bool**, 955  
**char**, 955  
enumerations, 956  
integer types, 955  
manipulating, 965–967  
signed, 961–965  
size, 955–956  
**unsigned**, 961–965  
**<bitset>**, 1133  
**bitset**, 959–961  
  bitwise logical operations, 960  
  construction, 959  
  exceptions, 1138  
  I/O, 960  
Bitwise logical operations, 956–959, 1094  
  and **&**, 956–957, 1089, 1094  
  or **|**, 956, 1089, 1094  
  or and assign, **|=**, 966  
  and and assign **&=**, 1090  
  complement **~**, 956  
  exclusive or **^**, 956, 1089, 1094  
  exclusive or and assign **^=**, 1089  
  left shift **<<**, 956  
  left shift and assign **<<=**, 1089  
  right shift **>>**, 956  
  right shift and assign **>>=**, 1089

Blackboard, 36  
 Black-box testing, 992–993  
**blank**, character class, `regex`, 878, 1179  
 Block, 111  
   debugging, 161  
   delimiter, 47, 111  
   nesting within functions, 271  
   **try** block, 146–147  
 Block comment `/*...*/`, 238  
 Blue marginal alerts, 3  
 BNF (Backus-Naur) Form, 823, 828  
 Body, functions, 114  
**bool**, 63, 66–67, 1099  
   bits in memory, 78  
   bit space, 955  
   C++ and C, 1026, 1038  
   size, 78  
**boolalpha**, manipulator, 1173  
 Boolean conversions, 1092  
 Borland, 831  
 Bottom-up approach, 9, 811  
 Bounds error, 149  
 Branching, testing, 1006–1008. *See also*  
   Conditional statements  
**break**, **case** label termination, 106–108  
 Broadcast functions, 903  
**bsearch()**, 1194–1195  
 Buffer, 348  
   flushing, 240–241  
**iostream**, 406  
   overflow, 661, 792, 1006. *See also* `gets()`, `scanf()`  
 Bugs, 158, 1218. *See also* Debugging; Testing  
   finding the last, 166–167  
   first documented, 824–825  
   regression tests, 993  
 Built-in types, 304, 1099  
   arrays, 747–749, 1101–1102  
**bool**, 77, 1100  
   characters, 77, 891, 1100  
   default constructors, 328  
   exceptions, 1126  
   floating-point, 77, 891–895, 1100  
   integers, 77, 891–895, 961–965, 1100  
   pointers, 588–590, 1100–1101  
   references, 279–280, 1102–1103  
**Button** example, 443, 561–563  
   attaching to menus, 571  
   detecting a click, 557  
 Byte, 78, 1218  
   operations, C-style strings, 1048–1049

## C

.**c** suffix, 1029  
.b.**pp**, suffix, 48, 1200  
 C# language, 831  
 C++ language, 839–842. *See also* Programming;  
   Programs; Software  
   coding standards, list of, 983  
   portability, 11  
   use for teaching, xxiv, 6–9  
 C++ and C, 1022–1024  
   C functions, 1028–1032  
   C linkage convention, 1033  
   C missing features, 1025–1027  
   calling one from the other, 1032–1034  
   casts, 1040–1041  
   compatibility, 1024–1025  
**const**, 1054–1055  
   constants, 1054–1055  
   container example, 1059–1065  
   definitions, 1038–1040  
**enum**, 1042  
   family tree, 1023  
   free-store, 1043–1045  
   input/output, 1050–1054  
   keywords, 1037–1038  
   layout rules, 1034  
   macros, 1054–1059  
**malloc()**, 1043–1044  
   namespaces, 1042–1043  
   nesting **structs**, 1037  
   old-style casts, 1040  
   opaque types, 1060  
   performance, 1024  
**realloc()**, 1045  
   structure tags, 1036–1037  
   type checking, 1032–1033  
**void**, 1030  
**void\***, 1041–1042  
 “C first” approach to programming, 9  
 C language, 836–839. *See also* C standard  
   library  
 C++ compatibility, 1022–1024. *See also*  
   C++ and C  
 K&R, 838, 1022–1023  
   linkage convention, 1033  
   missing features, 1025–1027  
 C standard library  
   C-style strings, 1191  
   header files, 1135

- input/output. *See* C-style I/O (stdio)  
memory, 1192–1193
- C-style casts, 1040–1041, 1087, 1095
- C-style I/O (stdio)
- %, conversion specification, 1187
  - conversion specifications, 1188–1189
  - file modes, 1186
  - files, opening and closing, 1186
  - fprintf()**, 1051–1052, 1187
  - getc()**, 1052, 1191
  - getchar()**, 1045, 1052–1053, 1191
  - gets()**, 1052, 1190–1191
  - output formats, user-defined types, 1189–1190
  - padding, 1188
  - printf()**, 1050–1051, 1187
  - scanf()**, 1052–1053, 1190
  - stderr**, 1189
  - stdin**, 1189
  - stdout**, 1189
  - truncation, 1189
- C-style strings, 654–655, 1045–1047, 1191
- byte operations, 1048–1049
  - const**, 1047–1048
  - copying, 1046–1047, 1049
  - executing as a command, **system()**, 1194
  - lexicographical comparison, 1046
  - operations, 1191–1192
  - pointer declaration, 1049–1050
  - strcat()**, concatenate, 1047
  - strchr()**, find character, 1048
  - strcmp()**, compare, 1046
  - strcpy()**, copy, 1047, 1049
  - from **string**, **c\_str()**, 350, 851
  - strlen()**, length of, 1046
  - strncat()**, 1047
  - strncmp()**, 1047
  - strncpy()**, 1047
  - three-way comparison, 1046
- CAD/CAM, 27, 34
- Calculator example, 174, 186–188
- analysis and design, 176–179
  - expression()**, 197–200
  - get\_token()**, 196
  - grammars and programming, 188–195
  - parsing, 190–193
  - primary()**, 196, 208
  - symbol table, 247
  - term()**, 196, 197–202, 206–207
- Token, 185–186
- Token\_stream**, 206–214, 240–241
- Call stack, 290
- Callback functions, 556–559
- Callback implementation, 1208–1209
- Calling functions. *See* Function calls
- calloc()**, 1193
- Cambridge University, 839
- capacity()**, 673–674, 1151
- Capital letters. *See* Case (of characters)
- Case (of characters)
- formatting, 397–398
  - identifying, 397
  - islower()**, 397, 1175
  - map** container, 782
  - in names, 74–77
  - sensitivity, 397–398
  - tolower()**, changing case, 398, 1176
  - toupper()**, changing case, 398, 1176
- case** labels, 106–108
- <cassert>**, 1135
- Casting away **const**, 609–610
- Casts. *See also* Type conversion
- C++ and C, 1026, 1038
  - casting away **const**, 609
  - const\_cast**, 1095
  - C-style casts, 1040–1041
  - dynamic\_cast**, 932, 1095
  - lexical\_cast** example, 855
  - narrow\_cast** example, 153
  - reinterpret\_cast**, 609
  - static\_cast**, 609, 944, 1095
  - unrelated types, 609
- CAT scans, 30
- catch**, 147, 1038
- Catch all exceptions ., 152
- Catching exceptions, 146–153, 239–241, 1126
- cb\_next()** example, 556–559
- <cctype>**, 1135, 1175
- ceil()**, 917, 1181
- cerr**, 151, 1169, 1189
- <cerrno>**, 1135
- <cfloat>**, 1135
- Chaining operations, 180–181
- Character classes
- list of, 1179
  - in regular expressions, 873–874, 878
- Character classification, 397–398, 1175–1176
- Character literals, 161, 1079–1080
- CHAR\_BIT** limit macro, 1181
- CHAR\_MAX** limit macro, 1181
- CHAR\_MIN** limit macro, 1181

**char** type, 63, 66–67, 78  
 bits, 955  
 built-in, 1099  
 properties, 741–742  
**signed** *vs.* **unsigned**, 894, 964  
**cin**, 61  
 C equivalent. *See* **stdin**  
 standard character input, 61, 347, 1169  
**Circle** example, 469–472, 497  
*vs.* **Ellipse**, 474  
 Circular reference. *See* Reference (circular)  
**class**, 183, 1036–1037  
 Class  
   abstract, 495, 512–513, 1118–1119. *See also*  
     Abstract classes  
   base, 504–506  
   coding standards, 981  
   concrete, 495–496, 1218  
   **const** member functions, 1110  
   constructors, 1112–1114, 1119–1120  
   copying, 1115, 1119  
   creating objects. *See* Concrete classes  
   default constructors, 327–330  
   defining, 212, 305, 1108, 1218  
   derived, 504  
   destructors, 1114–1115, 1119  
   encapsulation, 505  
   **friend** declaration, 1111  
   generated operations, 1119–1120  
   grouping related, 512  
   hierarchies, 512  
   history of, 834  
   implementation, 306–308  
   inheritance, 504–505, 513–514  
   interface, 513–514  
   member access. *See* Access control  
   naming. *See* Namespaces  
   nesting, 270  
   object layout, 506–507  
   organizing. *See* Namespaces  
   parameterized, 682–683. *See also* Template  
**private**, 306–308, 505, 511, 1108–1109  
**protected**, 495, 505, 511  
**public**, 306–308, 505, 511, 1108–1109  
 run-time polymorphism, 504–505  
 subclasses, 504. *See also* Derived classes  
 superclasses, 504. *See also* Base classes  
 templates, 681–683  
**this** pointer, 1110  
 types as parameters. *See* Template  
**union**, 1121  
 unqualified name, 1110  
 uses for, 305  
 Class interfaces, 323, 1108  
   argument types, 324–326  
   **const** member functions, 330–332  
   constants, 330–332. *See also* **const**  
   copying, 326–327  
   helper functions, 332–334  
   immutable values, 330–332  
   initializing objects, 327–330  
   members, 332–334  
   mutable values, 332–334  
   public *vs.* private, 306–308  
   symbolic constants, defining, 326  
   uninitialized variables, 327–330  
 Class members, 305, 1108  
   . (dot), 306, 1109  
   :: (scope resolution), 1109  
   accessing, 306. *See also* Access control  
   allocated at same address, 1121  
   bitfields, 1120–1121  
   in-class definition, 1112  
   class interfaces, 332–334  
   data, 305  
   definitions, 1112  
   function, 314–316  
   out-of-class definition, 1112  
**Token\_stream** example, 212  
**Token** example, 183–184  
 Class scope, 267, 1083  
 Class template  
   parameterized class, 682–683  
   parameterized type, 682–683  
   specialization, 681  
   type generators, 681  
**classic\_elimination()** example, 910–911  
 Cleaning up code  
   comments, 237–238  
   functions, 234–235  
   layout, 235–236  
   logical separations, 234–235  
   revision history, 237–238  
   scaffolding, 234–235  
   symbolic constants, 232–234  
**clear()**, 355–358, 1150  
**<climits>**, 1135  
**<locale>**, 1135

- clock()**, 1015–1016, 1193  
**clock\_t**, 1193  
**clone()** example, 504  
**Closed polyline** example, 456–458  
    vs. **Polygon**, 458  
**close()** file, 352  
**<cmath>**, 918, 1135, 1182  
**cntrl**, 878, 1179  
COBOL language, 823–825  
Code  
    definition, 1218  
    layout, cleaning up, 235–236  
    libraries, uses for, 177  
    storage, 591–592  
    structure, ideals, 810–811  
    test coverage, 1008  
Coding standards, 974–975  
    C++, list of, 983  
    complexity, sources of, 975  
    ideals, 976–977  
    sample rules, 977–983  
**Color** example, 425–426, 450–452  
    color chat example, 465–467  
    fill, 431–432, 462–464, 500  
    transparency, 451  
Columns, matrices, 900–901, 906  
Command-line, 47  
Comments, 45–46  
    block **/\* ... \*/**, 238, 1076  
    C++ and C, 1026  
    cleaning up, 237–238  
    vs. code, 238  
    line **//**, 45–46, 1076  
    role in debugging, 159–160  
Common Lisp language, 825  
Communication skills, programmers, 22  
Compacting garbage collection, 938–939  
Comparison, 67. *See also < : ==*  
    C-style strings, 1045–1047  
    characters, 740  
    containers, 1151  
    **key\_compare**, 1147  
    lexicographical, C-style strings, 1046  
    **lexicographical\_compare()**, 1162  
    **min/max** algorithms, 1161–1162  
    **string**, 851  
    three-way, 1046  
Compatibility. *See* C++ and C  
Compile-time errors. *See* Errors, compile-time  
Compiled languages, 47–48  
Compilers, 48, 1218  
    compile-time errors, 51  
    conditional compilation, 1058–1059  
    syntax checking, 48–50  
**compl**, synonym for **~**, 1037, 1082  
**complex**  
     $*$ , multiply, 919, 1183  
     $+$ , add (plus), 919, 1183  
     $\ll$ , output, 1183  
     $\neq$ , not equal (inequality), 919, 1183  
     $=$ , equal, 919, 1183  
     $\gg$ , input, 920, 1183  
     $/$ , divide, 919, 1183  
     $\ll$ , output, 920  
    **abs()**, absolute value, 920, 1183  
    **conj()**, conjugate, 920  
    Fortran language, 920  
    **imag()**, imaginary part, 920  
    **norm()**, square of **abs()**, 919  
    number types, 1182–1183  
    **polar()**, polar coordinate, 920  
    **real()**, real part, 920  
    rho, 920  
    square of **abs()**, 919  
    theta, 920  
**<complex>**, 1134  
    **complex** operators, 919–920, 1183  
    standard math functions, 1181  
Complex numbers, 919–920  
Complexity, 1218  
    sources of, 975  
Composite assignment operators, 73–74  
Compound statements, 111  
Computation, 91. *See also* Programs; Software  
    correctness, 92–94  
    data structures, 90–91  
    efficiency, 92–94  
    input/output, 91  
    objectives, 92–94  
    organizing programs, 92–94  
    programmer ideals, 92–94  
    simplicity, 92–94  
    state, definition, 90–91  
Computation *vs.* data, 717–720  
Computer-assisted surgery, 30  
Computers  
    CAT scans, 30  
    computer-assisted surgery, 30

- Computers, *continued*
 in daily life, 19–21
 information processing, 32
 Mars Rover, 33
 medicine, 30
 pervasiveness of, 19–21
 server farms, 31–32
 shipping, 26–28
 space exploration, 33
 telecommunications, 28–29
 timekeeping, 26
 world total, 19
 Computer science, 12, 24–25
 Concatenation of **strings**, 66
 **+**, 68–69, 851, 1176
 **+ =**, 68–69, 851, 1176
 Concept-based approach to programming, 6
 Concrete classes, 495–496, 1218
 Concrete-first approach to programming, 6
 Concurrency, 932
 Conditional compilation, 1058–1059
 Conditional expression **?:**, 268, 1089
 Conditional statements. *See also* Branching,
 testing
 **for**, 111–113
 **if**, 102–104
 **switch**, 105–109
 **while**, 109–111
 Conforming programs, 1075
 Confusing variable names, 77
 **conj()**, complex conjugate, 920, 1183
 Conjugate, 920
 Consistency, ideals, 814–815
 Console, as user interface, 552
 Console input/output, 552
 Console window, displaying, 162
 **const**, 95–97. *See also* Constant; Static storage,
 **static const**
 C++ and C, 1026, 1054–1055
 class interfaces, 330–332
 C-style strings, 1047–1048
 declarations, 262–263
 initializing, 262
 member functions, 330–332, 1110
 overloading on, 647–648
 passing arguments by, 276–278, 281–284
 type, 1099
 **\*const**, immutable pointer, 1099
 Constant. *See also* **const**, expressions, 1093
 **const\_cast**, casting away **const**, 609, 1095
 **const\_iterator**, 1147
 **constexpr**, 96–97, 290–291, 1093, 1104
 Constraints, **vector** range checking, 695
 Constructors, 310–312, 1112–1114. *See also*
 Destructors; Initialization
 containers, 1148
 copy, 633–634, 640–646
 **Date** example, 311
 **Date** example 307, 324–326
 debugging, 643–646
 default, 327–330, 1119
 error handling, 313, 700–702
 essential operations, 640–646
 exceptions, 700–702
 **explicit**, 642–643
 implicit conversions, 642–643
 initialization of bases and members, 315, 477, 555
 invariant, 313–314, 701–702
 move, 637–640
 need for default, 641
 **Token** example, 184
 Container adaptors, 1144
 Containers, 148, 749–751, 1218. *See also* Arrays;
 **list**; **map**, associative array; **vector**
 and algorithms, 722
 almost containers, 751, 1145
 assignments, 1148
 associative, 1144, 1151–1152
 **capacity()**, 1150–1151
 of characters. *See* **string**
 comparing, 1151
 constructors, 1148
 contiguous storage, 741
 copying, 1151
 destructors, 1148
 element access, 1149
 embedded systems, 951–954
 header files, 1133–1134
 information sources about, 750
 iterator categories, 752
 iterators, 1148
 list operations, 1150
 member types, 1147
 operations overview, 1146–1147
 queue operations, 1149
 sequence, 1144
 **size()**, 1150
 stack operations, 1149
 standard library, 1144–1152

- swapping, 1151  
templates, 686–687  
Contents of `*` (dereference, indirection), 594  
Contiguous storage, 741  
Control characters, `iscntrl()`, 397  
Control inversion, GUIs, 569–570  
Control variables, 110  
Controls. *See* `Widget` example  
Conversion specifications, `printf()`, 1188–1189  
Conversion. *See also* Type conversion  
    character case, 398  
    representation, 374–376  
    unchecked, 943–944  
Coordinates. *See also* `Point` example  
    computer screens, 419–420  
    graphs, 426–427  
`copy()`, 789–790, 1154  
Copy assignments, 634–636, 640–646  
Copy constructors, 633–634, 640–646  
`copy_backward()`, 1154  
`copy_if()`, 789  
Copying, 631–637  
    arrays, 653–654  
    class interfaces, 326–327  
    containers, 1151  
    C-style strings, 1046–1047, 1049  
    I/O streams, 790–793  
    objects, 503–504  
    sequences, 758, 789–794  
    `vector`, 631–636, 1148  
Correctness  
    definition, 1218  
    ideals, 92–94, 810  
    importance of, 929–930  
    software, 34  
`cos()`, cosine, 527–528, 917, 1181  
`cosh()`, hyperbolic cosine, 1182  
Cost, definition, 1219  
`count()`, 758, 1154  
`count_if()`, 758, 1154  
`cout`, 45  
    C equivalent. *See* `stdout`  
    printing error messages, 151. *See also* `cerr`  
    standard output, 347, 1169  
Critical systems, coding standards, 982–983  
`<cstddef>`, 1136  
`<cstdio>`, 1135  
`<cstdlib>`, 1135, 1193, 1194  
`c_str()`, 1177  
`<cstring>`, 1135, 1175, 1193  
`<ctime>`, 1135, 1193  
Ctrl D, 124  
Ctrl Z, 124  
Current object, 317. *See also* `this` pointer  
Cursor, definition, 45  
`<cwchar>`, 1136  
`<cwctype>`, 1136
- ## D
- `d`, any decimal digit, `regex`, 878, 1179  
`\d`, decimal digit, `regex`, 873, 1179  
`\D`, not a decimal digit, `regex`, 873, 1179  
`d` suffix, 1079  
Dahl, Ole-Johan, 833–835  
Data. *See also* Containers; Sequences; `list`; `map`,  
    associative array; `vector`  
abstraction, 816  
collections. *See* Containers  
    *vs.* computation, 717–720  
generalizing code, 714–716  
in memory. *See* Free store (heap storage)  
processing, overview, 712–716  
separating from algorithms, 722  
storing. *See* Containers  
structure. *See* Containers; `class`; `struct`  
traversing. *See* Iteration; Iterators  
uniform access and manipulation, 714–716. *See also* STL (Standard Template Library)  
Data member, 305, 492–493  
Data structure. *See* Data; `struct`  
Data type. *See* Type  
Date and time, 1193–1194  
**Date** example, *See* Chapters 6–7  
Deallocating memory, 598–600, 1094–1095. *See also* `delete[]`; `delete`  
Debugging, 52, 158, 1219. *See also* Errors; Testing  
    arrays and pointers, 656–659  
    assertions, 163  
    block termination, 161  
    bugs, 158  
    character literal termination, 161  
    commenting code, 159–160  
    compile-time errors, 161  
    consistent code layout, 160  
    constructors, 643–646  
    declaring names, 161  
    displaying the console window, 162  
    expression termination, 161  
    finding the last bug, 166–167

Debugging, *continued*  
 function size, 160  
 GUIs, 575–577  
 input data, 166  
 invariants, 162–163  
 keeping it simple, 160  
 logic errors, 154–156  
 matching parentheses, 161  
 naming conventions, 160  
 post-conditions, 165–166  
 pre-conditions, 163–165  
 process description, 158–159  
 reporting errors, 159  
 stepping through code, 162  
 string literal termination, 161  
 systematic approach, 166–167  
 test cases, 166, 227  
 testing, 1012  
 tracing code execution, 162–163  
 transient bugs, 595  
 using library facilities, 160  
 widgets, 576–577

**dec** manipulator, 382–383, 1174

Decimal digits, **isdigit()**, 397

Decimal integer literals, 1077

Decimal number system, 381–383, 1077–1078

Deciphering (decryption), example, 969–974

Declaration operators, 1099  
 & reference to, 276–279, 1099  
 \* pointer to, 587, 1099  
**I** array of, 649, 1099  
**O** function of, 113–115, 1099

Declarations, 51, 1098–1099  
 C++ and C, 1026  
 classes, 306  
 collections of. *See* Header files  
 constants, 262–263  
 definition, 51, 77, 257, 1098–1099, 1219  
*vs.* definitions, 259–260  
 entities used for, 261  
**extern** keyword, 259  
 forward, 261  
 function, 257–258, 1103  
 function arguments, 272–273  
 function return type, 272–273  
 grouping. *See* Namespaces  
 managing. *See* Header files  
 need for, 261  
 order of, 215

parts of, 1098  
 subdividing programs, 260–261  
 uses for, 1098  
 variables, 260, 262–263

Decrementing **--**, 97  
 iterator, 1141–1142  
 pointer, 652

Deep copy, 636

Default constructors, 328–329  
 alternatives for, 329–330  
 for built-in types, 328  
 initializing objects, 327  
 need for, identifying, 641  
 uses for, 328–329

**#define**, 1129

Definitions, 77, 258–259, 1219. *See also*  
 Declarations  
 C++ and C, 1038–1040  
*vs.* declarations, 259–260  
 function, 113–115, 272–273

**delete**  
 C++ and C, 1026, 1037  
 deallocating free store, 1094–1095  
 destructors, 601–605  
 embedded systems, 932, 936–940  
 free-store deallocation, 598–600  
 in unary expressions, 1087

**delete[]**, 599, 1087, 1094–1095

Delphi language, 831

Dependencies, testing, 1002–1003

Depth-first approach to programming, 6

**deque**, double ended queue, 1144

**<deque>**, 1133

Dereference/indirection  
 \*, 594. *See also* Contents of  
**I**, 118. *See also* Subscripting

Derivation, classes, 505

Derived classes, 505, 1219  
 access control, 511  
 base classes, 1116–1117  
 inheritance, 1116–1117  
 multiple inheritance, 1117  
 object layout, 506–507  
 overview, 504–506, 1116–1117

**private** bases and members, 511

**protected** bases and members, 511

**public** bases and members, 511  
 specifying, 507–508

virtual functions, 1117–1118

Design, 35, 176, 179, 1219  
Design for testing, 1011–1012  
Destructors, 601–603, 1114–1115, 1219. *See also Constructors*  
    containers, 1148  
    debugging, 643–646  
    default, 1119  
    essential operations, 640–646  
    exceptions, 700–702  
    freeing resources, 323, 700–702  
    and free store, 604–605  
    generated, 603  
    RAII, 700–702  
    virtual, 604–605  
    where needed, 641–642  
Device drivers, 346  
Dictionary examples, 123–125, 788  
**difference\_type**, 1147  
**digit**, character class, 878, 1179  
Digit, word origin, 1077  
Dijkstra, Edsger, 827–828, 992  
Dimensions, matrices, 898–901  
Direct expression of ideas, ideals, 811–812  
Dispatch, 504–505  
Display model, 413–414  
**distance()**, 1142  
Divide **/**, 66, 1088  
Divide and assign **/=**, 67, 1090  
Divide and conquer, 93  
Divide-by-zero error, 201–202  
**divides()**, 1164  
Domain knowledge, 934  
Dot product. *See inner\_product()*  
**double** floating-point type, 63, 66–67, 78, 1099  
Doubly-linked lists, 613, 725. *See also list*  
**draw()** example  
    fill color, 500  
    line visibility, 500  
    **Shape**, 500–502  
**draw\_lines()** example. *See also draw() example*  
    **Closed\_polyline**, 458  
    **Marked\_polyline**, 475–476  
    **Open\_polyline**, 456  
    **Polygon**, 459  
    **Rectangle**, 465  
    **Shape**, 500–502  
duration..., 1016, 1185  
**duration\_cast**, 1016, 1185

Dynamic dispatch, 504–505. *See also Virtual functions*  
Dynamic memory, 935–936, 1094. *See also Free store (heap storage)*  
**dynamic\_cast**, type conversion, 1095  
    exceptions, 1138  
    predictability, 932

## E

Efficiency  
    ideals, 92–94, 810  
    **vector** range checking, 695  
Einstein, Albert, 815  
Elements. *See also vector*  
    numbering, 649  
    pointers to, 650–652  
    variable number of, 649  
**Ellipse** example, 472–474  
    *vs. Circle*, 474  
Ellipsis ...  
    arguments (unchecked), 1105–1106  
    catch all exceptions, 152  
**else**, in **if**-statements, 102–104  
Email example, 855–865  
Embedded systems  
    coding standards, 975–977, 983  
    concurrency, 932  
    containers, 951–954  
    correctness, 929–930  
    **delete** operator, 932  
    domain knowledge, 934  
    **dynamic\_cast**, 932  
    error handling, 933–935  
    examples of, 926–928  
    exceptions, 932  
    fault tolerance, 930  
    fragmentation, 936, 937  
    free-store, 936–940  
    hard real time, 931  
    ideals, 932–933  
    maintenance, 929  
    memory management, 940–942  
    **new** operator, 932  
    predictability, 931, 932  
    real-time constraints, 931  
    real-time response, 928  
    reliability, 928  
    resource leaks, 931

Embedded systems, *continued*  
 resource limitations, 928  
 soft real time, 931  
 special concerns, 928–929

Empty  
**empty()**, is container empty? 1150  
 lists, 729  
 sequences, 729  
 statements, 101

Empty statement, 1035–1036

Encapsulation, 505

Enciphering (Encryption), example, 969–974

**end()**  
 iterator, 1148  
**string**, 851, 1177  
**vector**, 722

End of file  
**eof()**, 355, 1171  
 file streams, 366  
 I/O error, 355  
**stringstream**, 395

End of input, 124

End of line **\$** (in regular expressions), 873, 1178

Ending programs. *See* Termination

**endl** manipulator, 1174

**ends** manipulator, 1174

English grammar *vs.* programming grammar, 193–194

**enum**, 318–321, 1042. *See also* Enumerations

Enumerations, 318–321, 1107–1108  
**enum**, 318–321, 1042  
 enumerators, 318–321, 1107–1108

**EOF** macro, 1053–1054

**eof()** stream state, 355, 1171

**equal()**, 759, 1153

Equal **==**, 67, 1088

Equality operators, expressions, 1088

**equal\_range()**, 758, 796

**equal\_to()**, 1163

**erase()**  
**list**, 742–745, 1150  
 list operations, 615–617  
**string**, 851, 1177  
**vector**, 745–747

**errno**, error indicator, 918–919, 1182

**error()** example, 142–143  
 passing multiple strings, 152

Error diagnostics, templates, 683

Error handling. *See also* Errors; Exceptions  
**%** for floating-point numbers, 230–231  
 catching exceptions, 239–241

files fail to open, 389

GUIs, 576

hardware replication, 934

I/O errors. *See* I/O errors

I/O streams, 1171

mathematical errors, 918–919

modular systems, 934–935

monitoring subsystems, 935

negative numbers, 229–230

positioning in files, 393–394

predictable errors, 933

recovering from errors, 239–241

regular expressions, 878–880

resource leaks, 934

self-checking, 934

STL (Standard Template Library), 1137–1138

testing for errors, 225–229

transient errors, 934

**vector** resource exceptions, 702

Error messages. *See also* Reporting errors; **error()**  
 example; **runtime\_error**  
 exceptions, printing, 150–151  
 templates, 683  
 writing your own, 142

Errors, 1219. *See also* Debugging; Testing  
 classifying, 134  
 compile-time, 48–50, 134, 136–137  
 detection ideal, 135  
**error()**, 142–143  
 estimating results, 157–158  
 incomplete programs, 136  
 input format, 64–65  
 link-time, 134, 139–140  
 logic, 134, 154–156  
 poor specifications, 136  
 recovering from, 239–241. *See also* Exceptions  
 sources of, 136  
 syntax, 137–138  
 translation units, 139–140  
 type mismatch, 138–139  
 undeclared identifier, 258  
 unexpected arguments, 136  
 unexpected input, 136  
 unexpected state, 136

Errors, run-time, 134, 140–142. *See also*  
 Exceptions  
 callee responsibility, 143–145  
 caller responsibility, 142–143  
 hardware violations, 141  
 reasons for, 144–145  
 reporting, 145–146

- Essential operations, 640–646  
Estimating development resources, 177  
Estimating results, 157–158  
Examples  
    age distribution, 538–539  
    calculator. *See* Calculator example  
    **Date**. *See* **Date** example  
    deciphering, 969–974  
    deleting repeated words, 71–73  
    dictionary, 123–125, 788  
    Dow Jones tracking, 782–785  
    email analysis, 855–865  
    embedded systems, 926–928  
    enciphering (encryption), 969–974  
    exponential function, 527–528  
    finding largest element, 713–716, 723–724  
    fruits, 779–782  
    Gaussian elimination, 910–911  
    graphics, 414–418, 436  
    graphing data, 537–539  
    graphing functions, 527–528  
    GUI (graphical user interface), 565–569,  
        573–574, 576–577  
    Hello, World! 45–46  
    intrusive containers, 1059–1065  
    **Lines\_window**, 565–569, 573–574, 576–577  
    **Link**, 613–622  
    list (doubly linked), 613–622  
    **map** container, 779–785  
    **Matrix**, 908–914  
    palindromes, 659–662  
    **Pool** allocator, 940–941  
    **Punct\_stream**, 401–405  
    reading a single value, 359–363  
    reading a structured file, 367–376  
    regular expressions, 880–885  
    school table, 880–885  
    searching, 864–872  
    sequences, 723–724  
    **Stack** allocator, 942–943  
    TEA (Tiny Encryption Algorithm),  
        969–974  
    text editor, 734–741  
    **vector**. *See* **vector** example  
    **Widget** manipulation, 565–569,  
        1213–1216  
    windows, 565–569  
    word frequency, 777–779  
    writing a program. *See* Calculator example  
    writing files, 352–354  
    ZIP code detection, 864–872
- <exception>**, 1135  
Exceptions, 146–150, 1125–1126. *See also* Error  
    handling; Errors  
    bounds error, 149  
    C++ and C, 1026  
    **catch**, 147, 239–241, 1125–1126  
    **cerr**, 151–152  
    **cout**, 151–152  
    destructors, 1126  
    embedded systems, 932  
    error messages, printing, 150–151  
    exception, 152, 1138–1139  
    failure to catch, 153  
    GUIs, 576  
    input, 150–153  
    **narrow\_cast** example, 153  
    off-by-one error, 149  
    **out\_of\_range**, 149–150, 152  
    overview, 146–147  
    RAII (Resource Acquisition Is Initialization),  
        1125  
    range errors, 148–150  
    re-throwing, 702, 1126  
    **runtime\_error**, 142, 151, 153  
    stack unwinding, 1126  
    standard library exceptions, 1138–1139  
    terminating a program, 142  
    **throw**, 147, 1125  
    truncation, 153  
    type conversion, 153  
    uncaught exception, 153  
    user-defined types, 1126  
    **vector** range checking, 693–694  
    **vector** resources. *See* **vector**
- Executable code, 48, 1219  
Executing a program, 11, 1200–1201  
**exit()**, terminating a program, 1194–1195  
**explicit** constructor, 642–643, 1038  
Expression, 94–95, 1086–1090  
    coding standards, 980–981  
    constant expressions, 1093  
    conversions, 1091–1093  
    debugging, 161  
    grouping (**0**), 95, 867, 873, 876  
    lvalue, 94–95, 1090  
    magic constants, 96, 143, 232–234, 723  
    memory management, 1094–1095  
    mixing types, 99  
    non-obvious literals, 96  
    operator precedence, 95  
    operators, 97–99, 1086–1095

Expression, *continued*

- order of operations, 181
  - precedence, 1090
  - preserving values, 1091
  - promotions, 99, 1091
  - rvalue, 94–95, 1090
  - scope resolution, 1086
  - type conversion, 99–100, 1095
  - usual arithmetic conversions, 1092
- Expression statement, 100
- extern**, 259, 1033
- Extracting text from files, 856–861, 864–865

**F**

**f/F** suffix, 1079

- fail()** stream state, 355, 1171
- Falling through end of functions, 274
- false**, 1038
- Fault tolerance, 930
- fclose()**, 1053–1054, 1186
- Feature creep, 188, 201, 1219
- Feedback, programming, 36
- Fields, formatting, 387–388
- FILE**, 1053–1054
- File I/O, 349–350
- binary I/O, 391
  - close()**, 352
  - closing files, 352, 1186
  - converting representations, 374–376
  - modes, 1186
  - open()**, 352
  - opening files. *See* Opening files
  - positioning in files, 393–394
  - reading. *See* Reading files
  - writing. *See* Writing files
- Files, 1219. *See also* File I/O
- C++ and C, 1053–1054
- opening and closing, C-style I/O, 1186
- fill()**, 1157
- fill\_n()**, 1157
- Fill color example, 462–465, 500
- find()**, 758–761
- associative container operations, 1151
  - finding links, 615–617
  - generic use, 761–763
  - nonmodifying sequence algorithms, 1153
  - string operations, 851, 1177
- find\_end()**, 1153
- find\_first\_of()**, 1153

**find\_if()**, 758, 763–764

- Finding. *See also* Matching; Searching
- associative container operations, 1151
  - elements, 758
  - links, 615–617
  - patterns, 864–865, 869–872
  - strings, 851, 1177
- fixed** format, 387
- fixed** manipulator, 385, 1174
- <float.h>**, 894, 1181
- Floating-point, 63, 891, 1219
- % remainder (modulo), 201
  - assigning integers to, 892–893
  - assigning to integers, 893
  - conversions, 1092
  - fixed** format, 387
  - general** format, 387
  - input, 182, 201–202
  - integral conversions, 1091–1092
  - literals, 182, 1079
  - mantissa, 893
  - output, formatting, 384–385
  - precision, 386–387
  - and real numbers, 891
  - rounding, 386
  - scientific** format, 387
  - truncation, 893
- vector** example, 120–123
- float** type, 1099
- floor()**, 917, 1181
- FLTK (Fast Light Toolkit), 418, 1204
- code portability, 418
  - color, 451, 465–467
  - current style, obtaining, 500
  - downloading, 1204
  - fill, 465
  - in graphics code, 436
  - installing, 1205
  - lines, drawing, 454, 458
  - outlines, 465
  - rectangles, drawing, 465
  - testing, 1206
  - in Visual Studio, 1205–1206
  - waiting for user action, 559–560, 569–570
- flush** manipulator, 1174
- Flushing a buffer, 240–241
- Fonts for Graphics example, 468–470
- fopen()**, 1053–1054, 1186
- for**-statement, 111–113
- vs. **while**, 122

**for\_each()**, 119, 1153  
Ford, Henry, 806  
Formal arguments. *See* Parameters  
Formatting. *See also* C-style I/O; I/O streams;  
    Manipulators  
    *See also* C-style I/O, 1050–1054  
    *See also* I/O streams, 1172–1173  
    case, 397–398  
    *See also* Manipulators, 1173–1175  
    fields, 387–388  
    precision, 386–387  
    whitespace, 397  
Fortran language, 821–823  
    array indexing, 899  
    **complex**, 920  
    subscripting, 899  
Forward declarations, 261  
Forward iterators, 752, 1142  
**fprintf()**, 1051–1052, 1187  
Fragmentation, embedded systems, 936, 937  
**free()**, deallocate, 1043–1044, 1193  
Free store (heap storage)  
    allocation, 593–594  
    C++ and C, 1043–1045  
    deallocation, 598–600  
    **delete**, 598–600, 601–605  
    and destructors. *See* Destructors  
    embedded systems, 936–940  
    garbage collection, 600  
    leaks, 598–600, 601–605  
    **new**, 593–594  
    object lifetime, 1085  
Freeing memory. *See* Deallocating memory  
**friend**, 1038, 1111  
**from\_string()** example, 853–854  
**front()**, first element, 1149  
**front\_inserter()**, 1162  
**fstream()**, 1170  
**<fstream>**, 1134  
**fstream** type, 350–352  
Fully qualified names, 295–297  
**Function** example, 443, 525–528  
Function , 47, 113–117. *See also* Member functions  
    accessing class members, 1111  
    arguments. *See* Function arguments  
    in base classes, 504  
    body, 47, 114  
    C++ and C, 1028–1032  
    callback, GUIs, 556–559  
    calling, 1103  
    cleaning up, 234–235  
    coding standards, 980–981  
    common style, 490–491  
    debugging, 160  
    declarations, 117, 1103  
    definition, 113–115, 272, 1219  
    in derived classes, 501, 505  
    falling through, 274  
    formal arguments. *See* Function parameter  
        (formal argument)  
    **friend** declaration, 1111  
    generic code, 491  
    global variables, modifying, 269  
    graphing. *See* **Function** example  
    inline, 316, 1026  
    linkage specifications, 1106  
    naming. *See* Namespaces  
    nesting, 270  
    organizing. *See* Namespaces  
    overloading, 321–323, 526, 1026  
    overload resolution, 1104–1105  
    parameter, 115. *See also* Function parameter  
        (formal argument)  
    pointer to, 1034–1036  
    post-conditions, 165–166  
    pre-conditions, 163–165  
    pure virtual, 1221  
    requirements, 153. *See also* Pre-conditions  
    **return**, 113–115, 272–273, 1103  
    return type, 47, 272–273  
    standard mathematical, 528, 1181–1182  
    types as parameters. *See* Template  
    uses for, 115–116  
    **virtual**, 1034–1036. *See also* Virtual functions  
Function activation record, 287  
Function argument. *See also* Function parameter  
    (formal argument); Parameters  
    checking, 284–285  
    conversion, 284–285  
    declaring, 272–273  
    formal. *See* Parameters  
    naming, 273  
    omitting, 273  
    passing. *See* Function call  
Function call, 285  
    call stack, 290  
    **expression()** call example, 287–290  
    function activation record, 287  
    history of, 820  
    memory for, 591–592

- Function call, *continued*
- `0` operator, 766
  - pass by `const` reference, 276–278, 281–284
  - pass by non-`const` reference, 281–284
  - pass by reference, 279–284
  - pass by value, 276, 281–284
  - recursive, 289
  - stack growth, 287–290. *See also* Function
    - activation record
    - temporary objects, 282
  - Function-like macros, 1056–1058
  - Function member
    - definition, 305–306
    - same name as class. *See* Constructors
  - Function objects, 765–767
    - `0` function call operator, 766
    - abstract view, 766–767
    - adaptors, 1164
    - arithmetic operations, 1164
    - parameterization, 767
    - predicates, 767–768, 1163
  - Function parameter (formal argument)
    - `...` ellipsis, unchecked arguments, 1105–1106
    - pass by `const` reference, 276–278, 281–284
    - pass by non-`const` reference, 281–284
    - pass by reference, 279–284
    - pass by value, 276, 281–284
    - temporary objects, 282
    - unused, 272
  - Function template
    - algorithms, 682–683
    - argument deduction, 689–690
    - parameterized functions, 682–683
  - `<functional>`, 1133, 1163
  - Functional cast, 1095
  - Functional programming, 823
  - Fused multiply-add, 904
- ## G
- Gadgets. *See* Embedded systems
- Garbage collection, 600, 938–939
- Gaussian elimination, 910–911
- `gcount()`, 1172
- `general` format, 387
- `general` manipulator, 385
- `generate()`, 1157
- `generate_n()`, 1157
- Generic code, 491
- Generic programming, 682–683, 816, 1219
- Geometric shapes, 427
- `get()`, 1172
- `getc()`, 1052, 1191
- `getchar()`, 1053, 1191
- `getline()`, 395–396, 851, 855, 1172
- `gets()`, 1052
- C++ alternative `>>`, 1053
  - dangerous, 1052
  - `scanf()`, 1190
- `get_token()` example, 196
- GIF images, 480–482
- Global scope, 267, 270, 1082
- Global variables
  - functions modifying, 269
  - memory for, 591–592
  - order of initialization, 292–294
- Going out of scope, 268–269, 291
- `good()` stream state, 355, 1171
- GP. *See* Generic programming
- Grammar example
  - alternation, patterns, 194
  - English grammar, 193–194
  - `Expression` example, 197–200, 202–203
  - parsing, 190–193
  - repetition, patterns, 194
  - rules *vs.* tokens, 194
  - sequencing rules, 195
  - terminals. *See* Tokens
  - writing, 189, 194–195
- Graph example. *See also* Grids, drawing
- `Axis`, 424–426
  - coordinates, 426–427
  - drawing, 426–427
  - points, labeling, 474–476
- `Graph.h`, 421–422
- Graphical user interfaces. *See* GUIs (graphical user interfaces)
- Graphics, 412. *See also* Graphics example; `Color` example; `Shape` example
  - displaying, 479–482
  - display model, 413–414
  - drawing on screen, 423–424
  - encoding, 480
  - filling shapes, 431
  - formats, 480
  - geometric shapes, 427
  - GIF, 480–482
  - graphics libraries, 481–482
  - graphs, 426–427
  - images from files, 433–434
  - importance of, 412–413
  - JPEG, 480–482

- line style, 431  
loading from files, 433–434  
screen coordinates, 419–420  
selecting a sub-picture from, 480  
user interface. *See* GUIs (graphical user interfaces)
- Graphics example  
**Graph.h**, 421–422  
GUI system, giving control to, 423  
header files, 421–422  
**main()**, 421–422  
**Point.h**, 444  
points, 426–427  
**Simple\_window.h**, 444  
**wait\_for\_button()**, 423  
**Window.h**, 444
- Graphics example, design principles  
access control. *See* Access control  
**attach()** vs. **add()**, 491–492  
class diagram, 505  
class size, 489–490  
common style, 490–491  
data modification access, 492–493  
generic code, 491  
inheritance, interface, 513–514  
inheritances, implementation, 513–514  
mutability, 492–493  
naming, 491–492  
object-oriented programming, benefits of, 513–514  
operations, 490–491  
private data members, 492–493  
protected data, 492–493  
public data, 492–493  
types, 488–490  
width/height, specifying, 490
- Graphics example, GUI classes, 442–444. *See also* Graphics example, interfaces
- Button**, 443  
**In\_box**, 443  
**Menu**, 443  
**Out\_box**, 443  
**Simple\_window**, 422–424, 443  
**Widget**, 561–563, 1209–1210  
**Window**, 443, 1210–1212
- Graphics example, interfaces, 442–443. *See also* Graphics example, GUI classes
- Axis**, 424–426, 443, 529–532  
**Circle**, 469–472, 497  
**Closed\_polyline**, 456–458  
**Color**, 450
- Ellipse**, 472–474  
**Function**, 443, 524–528  
**Image**, 443, 479–482  
**Line**, 445–448  
**Line\_style**, 452–455  
**Lines**, 448–450, 497  
**Mark**, 478–479  
**Marked\_polyline**, 474–476  
**Marks**, 476–477, 497  
**Open\_polyline**, 455–456, 497  
**Point**, 426–427, 445  
**Polygon**, 427–428, 458–460, 497  
**Rectangle**, 428–431, 460–465, 497  
**Shape**, 444–445, 449, 493–494, 513–514  
**Text**, 431–433, 467–470
- Graphing data example, 538–546  
Graphing functions example, 520–524, 532–537
- Graph\_lib** namespace, 421–422  
**greater()**, 1163  
Greater than **>**, 67, 1088  
Greater than or equal **>=**, 1088  
**greater\_equal()**, 1163  
Green marginal alerts, 3  
Grids, drawing, 448–449, 452–455  
Grouping regular expressions, 867, 873, 876  
Guarantees, 701–702  
Guidelines. *See* Ideals  
GUIs (graphical user interfaces), 552–553. *See also* Graphics example, GUI classes  
callback functions, 556–559  
callback implementation, 1208–1209  
**cb\_next()** example, 556–559  
common problems, 575–577  
control inversion, 569–570  
controls. *See* Widget example  
coordinates, computer screens, 419–420  
debugging, 575–577  
error handling, 576  
examples, 565–569, 573–574, 576–577  
exceptions, 576  
FLTK (Fast Light Toolkit), 418  
layers of code, 557  
**next()** example, 558–559  
pixels, 419–420  
portability, 418  
standard library, 418–419  
toolkit, 418  
**vector\_ref** example, 1212–1213  
**vector** of references, simulating, 1212–1213

GUIs (graphical user interfaces), *continued*  
 wait loops, 559–560  
**wait\_for\_button()** example, 559–560  
 waiting for user action, 559–560,  
 569–570  
**Widget** example, 561–569, 1209–1210,  
 1213–1216  
**Window** example, 565–569, 1210–1212  
 GUI system, giving control to, 423

## H

.h file suffix, 46  
 Half open sequences, 119, 721  
 Hard real-time, 931, 981–982  
 Hardware replication, error handling, 934  
 Hardware violations, 141  
 Hashed container. *See* **unordered\_map**  
 Hash function, 785–786  
 Hashing, 785  
 Hash tables, 785  
 Hash values, 785  
 Header files, 46, 1219  
   C standard library, 1135–1136  
   declarations, managing, 264  
   definitions, managing, 264  
   graphics example, 421–422  
   including in source files, 264–266, 1129  
   multiple inclusion, 1059  
   standard library, 1133–1134  
 Headers. *See* Header files  
 Heap algorithm, 1160  
 Heap memory, 592, 935–936, 1084, 1160. *See also*  
   Free store (heap storage)  
 Hejlsberg, Anders, 831  
 “Hello, World!” program, 45–47  
 Helper functions  
   == equality, 333  
   != inequality, 333  
   class interfaces, 332–334  
**Date** example, 309–310, 332–333  
   namespaces, 333  
   validity checking date values, 310  
**hex** manipulator, 382–383, 1174  
 Hexadecimal digits, 397  
 Hexadecimal number system, 381–383,  
 1077–1078  
 Hiding information, 1220  
 Hopper, Grace Murray, 824–825  
 Hyperbolic cosine, **cosh()**, 918

Hyperbolic sine, **sinh()**, 918, 1182  
 Hyperbolic tangent, **tanh()**, 917

## I

I/O errors  
**bad()** stream state, 355  
**clear()**, 355–358  
 end of file, 355  
**eof()** stream state, 355  
 error handling, 1171  
**fail()** stream state, 355  
**good()** stream state, 355  
**ios\_base**, 357  
 recovering from, 355–358  
 stream states, 355  
 unexpected errors, 355  
**unget()**, 355–358  
 I/O streams, 1168–1169  
**>>** input operator, 855  
**<<** output operator, 855  
**cerr**, standard error output stream, 151–152,  
 1169, 1189  
**cin** standard input, 347  
 class hierarchy, 855, 1170–1171  
**cout** standard output, 347  
 error handling, 1171  
 formatting, 1172–1173  
**fstream**, 388–390, 393, 1170  
**get()**, 855  
**getline()**, 855  
 header files, 1134  
**ifstream**, 388–390, 1170  
 input operations, 1172  
 input streams, 347–349  
 iostream library, 347–349, 1168–1169  
**istream**, 347–349, 1169–1170  
**istringstream**, 1170  
**ofstream**, 388–390, 1170  
**ostream**, 347–349, 1168–1169  
**ostringstream**, 388–390, 1170  
 output operations, 1173  
 output streams, 347–349  
 standard manipulators, 382, 1173–1174  
 standard streams, 1169  
 states, 1171  
 stream behavior, changing, 382  
 stream buffers, **streambufs**, 1169  
 stream modes, 1170  
**string**, 855

**stringstream**, 395, 1170  
throwing exceptions, 1171  
unformatted input, 1172

IBM, 823

Ichbiah, Jean, 832

IDE (interactive development environment), 52

Ideals

- abstraction level, 812–813
- bottom-up approach, 811
- class interfaces, 323
- code structure, 810–811
- coding standards, 976–977
- consistency, 814–815
- correct approaches, 811
- correctness, 810
- definition, 1219
- direct expression of ideas, 811–812
- efficiency, 810
- embedded systems, 932–933
- importance of, 8
- KISS, 815
- maintainability, 810
- minimalism, 814–815
- modularity, 813–814
- overview, 808–809
- performance, 810
- software, 34–37
- on-time delivery, 810
- top-down approach, 811

Identifiers, 1081. *See also* Names

- reserved, 75–76. *See also* Keywords

**if**-statements, 102–104

**#ifdef**, 1058–1059

**#ifndef**, 1058–1059

**ifstream** type, 350–352

**imag()**, imaginary part, 920, 1183

**Image** example, 443, 479–482

Images. *See* Graphics

Imaginary part, 920

Immutable values, class interfaces, 330–332

Implementation, 1219

- class, 306–308
- inheritance, 513–514
- programs, 36

Implementation-defined feature, 1075

Implicit conversions, 642–643

**in** mode, 389, 1170

**In\_box** example, 443, 563–564

In-class member definition, 1112

**#include**, 46, 264–266, 1128–1129

Include guard, 1059

**includes()**, 1159

Including headers, 1129. *See also* **#include**

Incrementing **++**, 66, 721

- iterators, 721, 750, 1140–1141
- pointers, 651–652
- variables, 73–74, 97–98

Indenting nested code, 271

Inequality **!=** (not equal), 67, 1088, 1101

**complex**, 919, 1183

containers, 1151

helper function, 333

iterators, 721, 1141

**string**, 67, 851, 1176

Infinite loop, 1219

Infinite recursion, 198, 1220

Information hiding, 1220

Information processing, 32

Inheritance

- class diagram, 505
- definition, 504
- derived classes, 1116–1117
- embedded systems, 951–954
- history of, 834
- implementation, 513–514
- interface, 513–514
- multiple, 1117
- pointers *vs.* references, 612–613
- templates, 686–687

Initialization, 69–73, 1220

**{} initialization notation**, 83

arrays, 596–598, 654–656

constants, 262, 329–330, 1099

constructors, 310–312

**Date** example, 309–312

default, 263, 327, 1085

invariants, 313–314, 701–702

menus, 571

pointers, 596–598, 657

pointer targets, 596–598

**Token** example, 184

**initializer\_list**, 630

**inline**, 1037

Inline

- functions, 1026
- member functions, 316

**inner\_product()**, 759. *See also* Dot product

- description, 774–775
- generalizing, 775–776

**inner\_product()**, *continued*

- matrices, 904
- multiplying sequences, 1184
- standard library, 759, 770

**inplace\_merge()**, 1158

Input, 60–62. *See also* Input **>>**; I/O streams

- binary I/O, 390–393
- C++ and C, 1052–1053
- calculator example, 179, 182, 185, 201–202, 206–208
- case sensitivity, 64
- cin**, standard input stream, 61
- dividing functions logically, 359–362
- files. *See* File I/O
- format errors, 64–65
- individual characters, 396–398
- integers, 383–384
- istringstream**, 394
- line-oriented input, 395–396
- newline character **\n**, 61–62, 64
- potential problems, 358–363
- prompting for, 61, 179
- separating dialog from function, 362–363
- a series of values, 356–358
- a single value, 358–363
- source of exceptions, 150–153
- stringstream**, 395
- tab character **\t**, 64
- terminating, 61–62
- type sensitivity, 64–65
- whitespace, 64

Input **>>**, 61

- case sensitivity, 64
- complex**, 920, 1183
- formatted input, 1172
- multiple values per statement, 65
- strings, 851, 1177
- text input, 851, 855
- user-defined, 365
- whitespace, ignoring, 64

Input devices, 346–347

Input iterators, 752, 1142

Input loops, 365–367

Input/output, 347–349. *See also* Input; Output

- buffering, 348, 406
- C++ and C. *See* stdio
- computation overview, 91
- device drivers, 346
- errors. *See* I/O errors
- files. *See* File I/O
- formatting. *See* Manipulators; **printf()**

irregularity, 380

**istream**, 347–354

natural language differences, 406

**ostream**, 347–354

regularity, 380

streams. *See* I/O streams

strings, 855

text in GUIs, 563–564

whitespace, 397, 398–405

Input prompt **>**, 223

Inputs, testing, 1001

Input streams, 347–349. *See also* I/O streams

**insert()**

- list**, 615–617, 742–745
- map** container, 782
- string**, 851, 1150, 1177
- vector**, 745–747

**inserter()**, 1162

Inserters, 1162–1163

Inserting

- list** elements, 742–745
- into **strings**, 851, 1150, 1177
- vector** elements, 745–747

Installing

- FLTK (Fast Light Toolkit), 1205
- Visual Studio, 1198

Instantiation, templates, 681, 1123–1124

**int**, integer type, 66–67, 78, 1099

- bits in memory, 78, 955

Integers, 77–78, 890–891, 1220

- assigning floating-point numbers to, 893
- assigning to floating-point numbers, 892–893
- decimal, 381–383
- input, formatting, 383–384
- largest, finding, 917
- literals, 1077
- number bases, 381–383
- octal, 381–383
- output, formatting, 381–383
- reading, 383–384
- smallest, finding, 917

Integral conversions, 1091–1092

Integral promotion, 1091

Interactive development environment (IDE), 52

Interface classes. *See* Graphics example, interfaces

Interfaces, 1220

- classes. *See* Class interfaces
- inheritance, 513–514
- user. *See* User interfaces

**internal** manipulator, 1174

Intrusive containers, example, 1059–1065

- Invariants, 313–314, 1220. *See also* Post-conditions;  
    Pre-conditions  
    assertions, 163  
    **Date** example, 313–314  
    debugging, 162–163  
    default constructors, 641  
    documenting, 815  
    invention of, 828  
    **Polygon** example, 460  
Invisible. *See* Transparency  
**<iomanip>**, 1134, 1173  
**<ios>**, 1134, 1173  
**<iostfwd>**, 1134  
**iostream**  
    buffers, 406  
    C++ and C, 1050  
    exceptions, 1138  
    library, 347–349  
**<iostream>**, 1134, 1173  
Irregularity, 380  
**is\_open()**, 1170  
**isalnum()** classify character, 397, 1175  
**isalpha()** classify character, 247, 397, 1175  
**iscntrl()** classify character, 397, 1175  
**isdigit()** classify character, 397, 1175  
**isgraph()** classify character, 397, 1175  
**islower()** classify character, 397, 1175  
**isprint()** classify character, 397, 1175  
**ispunct()** classify character, 397, 1175  
**isspace()** classify character, 397, 1175  
**istream**, 347–349, 1169–1170  
    **>>**, text input, 851, 1172  
    **>>**, user-defined, 365  
    binary I/O, 390–393  
    connecting to input device, 1170  
    file I/O, **fstream**, 349–354, 1170  
    **get()**, get a single character, 397  
    **getline()**, 395–396, 1172  
    **stringstreams**, 395  
    unformatted input, 395–396, 1172  
        using together with stdio, 1050  
**<istream>**, 1134, 1168–1169, 1173  
**istream\_iterator** type, 790–793  
**istringstream**, 394  
**isupper()** classify character, 397, 1175  
**isxdigit()** classify character, 397, 1175  
Iteration. *See also* Iterators  
    control variables, 110  
    definition, 1220  
    example, 737–741  
    linked lists, 727–729, 737–741  
    loop variables, 110–111  
    **for**-statements, 111–113  
    strings, 851  
    through values. *See* **vector**  
    **while**-statements, 109–111  
**iterator**, 1147  
**<iterator>**, 1133, 1162  
Iterators, 721–722, 1139–1140, 1220. *See also* STL  
    iterators  
        bidirectional iterator, 752  
        category, 752, 1142–1143  
        containers, 1143–1145, 1148  
        empty list, 729  
        example, 737–741  
        forward iterator, 752  
        header files, 1133–1134  
        input iterator, 752  
        operations, 721, 1141–1142  
        output iterator, 752  
        *vs.* pointers, 1140  
        random-access iterator, 752  
        sequence of elements, 1140–1141  
    **iter\_swap()**, 1157
- ## J
- Japanese age distribution example, 538–539  
JPEG images, 480–482
- ## K
- Kernighan, Brian, 838–839, 1022–1023  
**key\_comp()**, 1152  
**key\_compare**, 1147  
**key\_type**, 1147  
Key, value pairs, containers for, 776  
Keywords, 1037–1038, 1081–1082  
KISS, 815  
Knuth, Don, 808  
K&R, 838, 1022
- ## L
- I/L** suffix, 1077  
**\l**, “lowercase character,” **regex**, 873, 1179  
**\L**, “not lowercase character,” **regex**, 874, 1179  
Label  
    access control, 306, 511  
    **case**, 106–108  
    graph example, 529–532  
    of statement, 1096

Lambda expression, 560–561  
 Largest integer, finding, 917  
 Laws of optimization, 931  
 Layers of code, GUIs, 557  
 Layout rules, 979, 1034  
 Leaks, memory, 598–600, 601–605, 937  
 Leap year, 309  
**left** manipulator, 1174  
 Legal programs, 1075  
**length()**, 851, 1176  
 Length of strings, finding, 851, 1046, 1176  
**less()**, 1163  
 Less than **<**, 1088  
 Less than or equal **<=**, 67, 1088  
**less\_equal()**, 1163  
 Letters, identifying, 247, 397  
**lexical\_cast**, 855  
 Lexicographical comparison  
   **<=** comparison, 1176  
   **<** comparison, 1176  
   **>=** comparison, 1176  
   **>** comparison, 1176  
   **<** comparison, 851  
 C-style strings, 1046  
**lexicographical\_compare()**, 1162  
 Libraries, 51, 1220. *See also* Standard library  
   role in debugging, 160  
   uses for, 177  
 Lifetime, objects, 1085–1086, 1220  
 Limit macros, 1181  
**<limits>**, 894, 1135, 1180  
 Limits, 894–895  
**<limits.h>**, 894, 1181  
 Linear equations example, 908–914  
   **back\_substitution()**, 910–911  
   **classic\_elimination()**, 910–911  
   Gaussian elimination, 910–911  
   pivoting, 911–912  
   testing, 912–914  
 Line comment **//**, 45  
**Line** example, 445–447  
   *vs.* **Lines**, 448  
 Line-oriented input, 395–396  
**Lines** example, 448–450, 497  
   *vs.* **Line**, 448  
 Lines (graphic), drawing. *See also* Graphics;  
   **draw\_lines()**  
   on graphs, 529–532  
   line styles, 452–455  
   multiple lines, 448–450  
   single lines, 445–447  
   styles, 431, 454  
   visibility, 500  
 Lines (of text), identifying, 736–737  
**Line\_style** example, 452–455  
**Lines\_window** example, 565–569, 573–574, 576–577  
**Link** example, 613–622  
 Link-time errors. *See* Errors, link-time  
 Linkage convention, C, 1033  
 Linkage specifications, 1106  
 Linked lists, 725. *See also* Lists  
 Linkers, 51, 1220  
 Linking programs, 51  
 Links, 613–615, 620–622, 725  
 Lint, consistency checking program, 836  
 Lisp language, 825–826  
**list**, 727, 1146–1151  
   {} initialization notation, 83  
   **add()**, 615–617  
   **advance()**, 615–617  
   **back()**, 737  
   **erase()**, 615–617, 742–745  
   **find()**, 615–617  
   **insert()**, 615–617, 742–745  
   operations, 615–617  
   properties, 741–742  
   referencing last element, 737  
   sequence containers, 1144  
   subscripting, 727  
**<list>**, 1133  
 Lists  
   containers, 1150  
   doubly linked, 613, 725  
   empty, 729  
   erasing elements, 742–745  
   examples, 613–615, 734–741  
   finding links, 615–617  
   getting the *n*th element, 615–617  
   inserting elements, 615–617, 742–745  
   iteration, 727–729, 737–741  
   link manipulation, 615–617  
   links, examples, 613–615, 620–622, 726  
   operations, 726–727  
   removing elements, 615–617  
   singly linked, 612–613, 725  
   **this** pointer, 618–620  
 Literals, 62, 1077, 1220  
   character, 161, 1079–1080  
   decimal integer, 1077  
   in expressions, 96  
   **f/F** suffix, 1079  
   floating-point, 1079

- hexadecimal integer, 1077  
integer, 1077  
**I/L** suffix, 1077  
magic constants, 96, 143, 232–234, 723  
non-obvious, 96  
null pointer, **0**, 1081  
number systems, 1077–1079  
octal integer, 1077  
special characters, 1079–1080  
string, 161, 1080  
termination, debugging, 161  
for types, 63  
**u/U** suffix, 1077  
unsigned, 1077  
Local (automatic) objects, lifetime, 1085  
Local classes, nesting, 270  
Local functions, nesting, 270  
Local scope, 267, 1083  
Local variables, array pointers, 658  
Locale, 406  
**<locale>**, 1135  
**log()**, 918, 1182  
**log10()**, 918, 1182  
Logic errors. *See* Errors, logic  
Logical and **&&**, 1089, 1094  
Logical operations, 1094  
Logical or **||**, 1089, 1094  
**logical\_and()**, 1163  
**logical\_not()**, 1163  
**logical\_or()**, 1163  
Logs, graphing, 528  
**long** integer, 955, 1099  
Look-ahead problem, 204–209  
Loop, 110–111, 112, 1220  
    examples, parser, 200  
    infinite, 198, 1219  
    testing, 1005–1006  
    variable, 110–111, 112  
Lovelace, Augusta Ada, 832  
**lower**, 878, 1179  
**lower\_bound()**, 796, 1152, 1158  
Lower case. *See* Case (of characters)  
Lucent Bell Labs, 838  
Lvalue, 94–95, 1090
- M**
- Machine code. *See* Executable code  
Macros, 1055–1056  
    conditional compilation, 1058–1059  
**#define**, 1056–1058, 1129  
function-like, 1056–1058  
**#ifdef**, 1058–1059  
**#ifndef**, 1059  
**#include**, 1058, 1128–1129  
include guard, 1059  
naming conventions, 1055  
syntax, 1058  
uses for, 1056  
Macro substitution, 1129  
Maddock, John, 865  
Magic constants, 96, 143, 232–234, 723  
Magical approach to programming, 10  
**main()**, 46–47  
    arguments to, 1076  
    global objects, 1076  
    return values, 47, 1075–1076  
    starting a program, 1075–1076  
Maintainability, software, 35, 810  
Maintenance, 929  
**make\_heap()**, 1160  
**make\_pair()**, 782, 1165–1166  
**make\_unique()**, 1167  
**make\_vec()**, 702  
**malloc()**, 1043–1044, 1193  
Manipulators, 382, 1173–1174  
    complete list of, 1173–1174  
**dec**, 1174  
**endl**, 1174  
**fixed**, 1174  
**hex**, 1174  
**noskipws**, 1174  
**oct**, 1174  
**resetiosflags()**, 1174  
**scientific**, 1174  
**setiosflags()**, 1174  
**setprecision()**, 1174  
**skipws**, 1174  
Mantissa, 893  
**map**, associative array, 776–782. *See also* **set**, **unordered\_map**  
**I**, subscripting, 777, 1151  
balanced trees, 780–782  
binary search trees, 779  
case sensitivity, **No\_case** example, 795  
counting words example, 777–779  
Dow Jones example, 782–785  
email example, 855–872  
**erase()**, 781, 1150  
finding elements in, 776–777, 781, 1151–1152  
fruits example, 779–782

**map**, associative array, *continued*

- insert()**, 782, 1150
- iterators, 1144
- key storage, 776
- make\_pair()**, 782
- No\_case** example, 782, 795
- Node** example, 779–782
- red-black trees, 779
- vs.* **set**, 788
- standard library, 1146–1152
- tree structure, 779–782
- without values. *See set*
- <map>**, 776, 1133
- mapped\_type**, 1147
- Marginal alerts, 3
- Mark** example, 478–479
- Marked\_polyline** example, 474–476
- Marks** example, 476–477, 497
- Mars Rover, 33
- Matching. *See also* Finding; Searching
  - regular expressions, **regex**, 1177–1179
  - text patterns. *See* Regular expressions
- Math functions, 528, 1181–1182
- Mathematics. *See* Numerics
- Mathematical functions, standard
  - abs()**, absolute value, 917
  - acos()**, arccosine, 917
  - asin()**, arcsine, 918
  - atan()**, arctangent, 918
  - ceil()**, 917
  - <cmath>**, 918, 1135
  - <complex>**, 919–920
  - cos()**, cosine, 917
  - cosh()**, hyperbolic cosine, 918
  - errno**, error indicator, 918–919
  - error handling, 918–919
  - exp()**, natural exponent, 918
  - floor()**, 917
  - log()**, natural logarithm, 918
  - log10()**, base-10 logarithm, 918
  - sin()**, sine, 917
  - sinh()**, hyperbolic sine, 918
  - sqrt()**, square root, 917
  - tan()**, tangent, 917
  - tanh()**, hyperbolic tangent, 917
- Matrices, 899–901, 905–906
- Matrix** library example, 899–901, 905
  - I**, subscripting (C style), 897, 899
  - O**, subscripting (Fortran style), 899
  - accessing array elements, 899–901
- apply()**, 903
- broadcast functions, 903
- clear\_row**, 906
- columns, 900–901, 906
- dimensions, 898–901
- dot product, 904
- fused multiply-add, 904
- initializing, 906
- inner\_product**, 904
- input/output, 907
- linear equations example, 910–914
- multidimensional matrices, 898–908
- rows, 900–901, 906
- scale\_and\_add()**, 904
- slice()**, 901–902, 905
- start\_row**, 906
- subscripting, 899–901, 905
- swap\_columns()**, 906
- swap\_rows()**, 906
- max()**, 1161
- max\_element()**, 1162
- max\_size()**, 1151
- McCarthy, John, 825–826
- McIlroy, Doug, 837, 1032
- Medicine, computer use, 30
- Member, 305–307. *See also* Class
  - allocated at same address, 1121
  - class, nesting, 270
  - in-class definition, 1112
  - definition, 1108
  - definitions, 1112
  - out-of-class definition, 1112
- Member access. *See also* Access control
  - .** (dot), 1109
  - ::** scope resolution, 315, 1109
  - notation, 184
  - operators, 608
  - this** pointer, 1110
  - by unqualified name, 1110
- Member function. *See also* Class members;
  - Constructors; Destructors; **Date** example
  - calls, 120
  - nesting, 270
  - Token** example, 184
- Member initializer list, 184
- Member selection, expressions, 1087
- Member types
  - containers, 1147
  - templates, 1124
- memchr()**, 1193

**memcmp()**, 1192  
**memcpy()**, 1192  
**mem\_fn()** adaptor, 1164  
**memmove()**, 1192  
Memory, 588–590  
    addresses, 588  
    allocating. *See Allocating memory*  
    automatic storage, 591–592  
    **bad\_alloc** exception, 1094  
    for code, 591–592  
    C standard library functions, 1192–1193  
    deallocating, 598–600  
    embedded systems, 940–942  
    exhausting, 1094  
    freeing. *See Deallocation memory*  
    free store, 592–594  
    for function calls, 591–592  
    for global variables, 591–592  
    heap. *See Free store (heap storage)*  
    layout, 591–592  
    object layout, 506–507  
    object size, getting, 590–591  
    pointers to, 588–590  
    **sizeof**, 590–591  
    stack storage, 591–592  
    static storage, 591–592  
    text storage, 591–592  
**<memory>**, 1134  
**memset()**, 1193  
**Menu** example, 443, 564–565, 570–575  
**merge()**, 758, 1158  
Messages to the user, 564  
**min()**, 1161  
**min\_element()**, 1162  
Minimalism, ideals, 814–815  
**minus()**, 1164  
Missing copies, 645  
MIT, 825–826, 838  
Modifying sequence algorithms, 1154–1156  
Modularity, ideals, 813–814  
Modular systems, error handling, 934–935  
Modulo (remainder) **%**, 66. *See also* Remainder  
**modulus()**, 1164  
Monitoring subsystems, error handling, 935  
**move()**, 502, 562  
Move assignments, 637–640  
Move backward **-=**, 1101  
Move forward **+=**, 1101  
Move constructors, 637–640  
Moving, 637–640  
Multi-paradigm programming languages, 818  
Multidimensional matrices, 898–908  
**multimap**, 776, 860–861, 1144  
**<multimap>**, 776  
Multiplicative operators, expressions, 1088  
**multiples()**, 1164  
Multiply **\***, 66, 1088  
Multiply and assign **\*=**, 67  
**multiset**, 776, 1144  
**<multiset>**, 776  
Mutability, 492–493, 1220  
    class interfaces, 332–334  
    and copying, 503–504  
**mutable**, 1037  
Mutating sequence algorithms, 1154–1156

## N

**\n** newline, character literal, 61–62, 64, 1079  
Named character classes, in regular expressions, 877–878  
Names, 74–77  
    \_ (underscore), 75, 76  
    capital letters, 76–77  
    case sensitivity, 75  
    confusing, 77  
    conventions, 74–75  
    declarations, 257–258  
    descriptive, 76  
    function, 47  
    length, 76  
    overloaded, 140, 508–509, 1104–1105  
    reserved, 75–76. *See also* Keywords  
**namespace**, 271, 1037  
Namespaces, 294, 1127. *See also* Scope  
    `::` scope resolution, 295–296  
    C++ and C, 1042–1043  
    fully qualified names, 295–297  
    helper functions, 333  
    objects, lifetime, 1085  
    scope, 267, 1082  
    **std**, 296–297  
    for the STL, 1136  
    **using** declarations, 296–297  
    **using** directives, 296–297, 1127  
    variables, order of initialization, 292–294  
Naming conventions, 74–77  
    coding standards, 979–980  
    functions, 491–492  
    macros, 1055

Naming conventions, *continued*  
 role in debugging, 160  
 scope, 269  
**narrow\_cast** example, 153  
 Narrowing conversions, 80–83  
 Narrowing errors, 153  
 Natural language differences, 406  
 Natural logarithms, 918  
 Naur, Peter, 827–828  
**negate()**, 1164  
 Negative numbers, 229–230  
 Nested blocks, 271  
 Nested classes, 270  
 Nested functions, 270  
 Nesting  
   blocks within functions, 271  
   classes within classes, 270  
   classes within functions, 270  
   functions within classes, 270  
   functions within functions, 271  
   indenting nested code, 271  
   local classes, 270  
   local functions, 271  
   member classes, 270  
   member functions, 270  
   **structs**, 1037  
**new**, 592, 596–598  
 C++ and C, 1026, 1037  
 and **delete**, 1094–1095  
 embedded systems, 932, 936–940  
 example, 593–594  
 exceptions, 1138  
 types, constructing, 1087  
**<new>**, 1135  
 New-style casts, 1040  
**next\_permutation()**, 1161  
 No-throw guarantee, 702  
**noboolalpha**, 1173  
**No\_case** example, 782  
**Node** example, 779–782  
 Non-algorithms, testing, 1001–1008  
 Non-errors, 139  
 Non-intrusive containers, 1059  
 Nonmodifying sequence algorithm, 1153–1154  
 Non-narrowing initialization, 83  
 Nonstandard separators, 398–405  
**norm()**, 919, 1183  
 Norwegian Computing Center, 833–835  
**noshowbase**, 383, 1173  
**noshowpoint**, 1173  
**noshowpos**, 1173  
**noskipws**, 1174  
**not**, synonym for **!** 1037, 1038  
 Not **!** 1087  
**not1()** adaptor, 1164  
**not2()** adaptor, 1164  
 Notches, graphing data example, 529–532, 543–546  
 Not-conforming constructs, 1075  
 Not equal **!=** (inequality), 67, 1088, 1101  
**not\_eq**, synonym for **!=**, 1038  
**not\_equal\_to()**, 1163  
**nouppercase** manipulator, 1174  
**now()**, 1016, 1185  
**nth\_element()**, 1158  
 Null pointer, 598, 656–657, 1081  
**nullptr**, 598  
**Number** example, 189  
 Number systems  
   base-2, binary, 1078–1079  
   base-8, octal, 381–384, 1077–1078  
   base-10, decimal, 381–384, 1077–1078  
   base-16, hexadecimal, 381–384, 1077–1078  
**<numeric>**, 1135, 1183  
 Numerical algorithms. *See* Algorithms, numerical  
 Numerics, 890–891  
   absolute values, 917  
   arithmetic function objects, 1164  
   arrays. *See* **Matrix** library example  
   **<cmath>**, 918  
   columns, 895–896  
   **complex**, 919–920, 1182–1183  
   **<complex>**, 919–920  
   floating-point rounding errors, 892–893  
   header files, 1134  
   integer and floating-point, 892–893  
   integer overflow, 891–893  
   largest integer, finding, 917  
   limit macros, 1181  
   limits, 894  
   mantissa, 893  
   mathematical functions, 917–918  
   **Matrix** library example, 897–908  
   multi-dimensional array, 895–897  
   **numeric\_limits**, 1180  
   numerical algorithms, 1183–1184  
   overflow, 891–895  
   precision, 891–895  
   random numbers, 914–917  
   real numbers, 891. *See also* Floating-point

results, plausibility checking, 891  
rounding errors, 891  
rows, 895–896  
size, 891–895  
**sizeof()**, 892  
smallest integer, finding, 917  
standard mathematical functions, 917–918,  
    1181–1182  
truncation, 893  
**valarray**, 1183  
whole numbers. *See* Integers  
Nygaard, Kristen, 833–835

## O

.**obj** file suffix, 48  
Object, 60, 1220  
    aliases. *See* References  
    behaving like a function. *See* Function object  
    constructing, 184  
    copying, 1115, 1119  
    current (**this**), 317  
**Date** example, 334–338  
initializing, 327–330. *See also* Constructors  
layout in memory, 308–309, 506–507  
lifetime, 1085–1086  
named. *See* Variables  
**Shape** example, 495  
**sizeof()**, 590–591  
state, 2, 305  
type, 77–78  
value. *See* Values  
Object code, 48, 1220. *See also* Executable code  
Object-oriented programming, 1220  
    “from day one,” 10  
    *vs.* generic programming, 682  
    for graphics, benefits of, 513–514  
    history of, 816, 834  
**oct** manipulator, 382–383, 1174  
Octal number system, 381–383, 1077–1078  
Off-by-one error, 149  
**ofstream**, 351–352  
Old-style casts, 1040  
One-dimensional (1D) matrices, 901–904  
On-time delivery, ideals, 810  
**\ooo** octal, character literal, 1080  
OOP. *See* Object-oriented programming  
Opaque types, 1060  
**open()**, 352, 1170  
Open modes, 389–390

Open shapes, 455–456  
Opening files, 350–352. *See also* File I/O  
    binary files, 390–393  
    **binary** mode, 389  
    C-style I/O, 1186  
    failure to open, 389  
    file streams, 350–352  
    nonexistent files, 389  
    open modes, 389–390  
    testing after opening, 352  
**Open polyline** example, 455–456, 497  
Operations, 66–69, 305, 1220  
    chaining, 180–181  
    graphics classes, 490–491  
**operator**, 1038  
Operator overloading, 321  
    C++ standard operators, 322–323  
    restrictions, 322  
    user-defined operators, 322  
    uses for, 321–323  
Operator, 97–99  
    != not, 1087  
    != not-equal (inequality), 1088  
    & (unary) address of, 588, 1087  
    & (binary) bitwise and, 956, 1089, 1094  
    && logical and, 1089, 1094  
    &= and and assign, 1090  
    % remainder (modulo), 1088  
    %= remainder (modulo) and assign, 1090  
    \* (binary) multiply, 1088  
    \* (unary) object contents, pointing to, 1087  
    \*= multiply and assign, 1089  
    + add (plus), 1088  
    ++ increment, 1087  
    += add and assign, 1090  
    - subtract (minus), 65, 1088  
    -- decrement, 66, 1087, 1141  
    -> (arrow) member access, 608, 1087, 1109,  
        1141  
    . (dot) member access, 1086–1087  
    / divide, 1088  
    /= divide and assign, 1090  
    :: scope resolution, 1086  
    < less than, 1088  
    << shift left, 1088. *See also* **ostream**  
    <<= shift left and assign, 1090  
    <= less than or equal, 1088  
    = assign, 1089  
    == equal, 1088  
    > greater than, 1088

Operator, *continued*

**>=** greater than or equal, 1088  
**>>** shift right, 1088. *See also istream*  
**>>=** shift right and assign, 1090  
**?:** conditional expression (arithmetic if), 1089  
**[ ] subscript**, 1086  
**^** bitwise exclusive or, 1089, 1094  
**^=** xor and assign, 1090  
**|** bitwise or, 1089, 1094  
**|=** or and assign, 1090  
**||** logical or, 1089, 1094  
**~** complement, 1087  
 additive operators, 1088  
**const\_cast**, 1086, 1095  
**delete**, 1087, 1094–1095  
**delete[]**, 1087, 1094–1095  
 dereference. *See* Contents of  
**dynamic\_cast**, 1086, 1095  
 expressions, 1086–1095  
**new**, 1087, 1094–1095  
**reinterpret\_cast**, 1086, 1095  
**sizeof**, 1087, 1094  
**static\_cast**, 1086, 1095  
**throw**, 1090  
**typeid**, 1086  
 Optimization, laws of, 931  
**or**, synonym for **|**, 1038  
 Order of evaluation, 291–292  
**or\_eq**, synonym for **|=**, 1038  
**ostream**, 347–349, 1168–1169  
     **<<**, text output, 851, 855  
     **<<**, user-defined, 363–365  
 binary I/O, 390–393  
 connecting to output device, 1170  
 file I/O, **fstream**, 349–354, 1170  
**stringstreams**, 395  
 using together with stdio, 1050  
**<ostream>**, 1134, 1168–1169, 1173  
**ostream\_iterator** type, 790–793  
**ostringstream**, 394–395  
**out** mode, 389, 1170  
 Out-of-class member definition, 1112  
 Out-of-range conditions, 595–596  
**Out\_box** example, 443, 563–564  
**out\_of\_range**, 149–150, 152  
 Output, 1220. *See also* Input/output; I/O streams  
     devices, 346–347  
     to file. *See* File I/O, writing files  
     floating-point values, 384–385

format specifier **%**, 1187

formatting. *See* Input/output, formatting

integers, 381–383

iterator, 752, 1142

operations, 1173

streams. *See* I/O streams

to string. *See* stringstream

testing, 1001

Output **<<**, 47, 67, 1173

**complex**, 920, 1183

**string**, 851

text output, 851, 855

user-defined, 363–365

Overflow, 891–895, 1220

Overloading, 1104–1105, 1221

alternative to, 526

C++ and C, 1026

on **const**, 647–648

linkage, 140

operators. *See* Operator overloading

and overriding, 508–511

resolution, 1104–1105

Override, 508–511, 1221

## P

Padding, C-style I/O, 1188

**pair**, 1165–1166

reading sequence elements, 1152–1153

searching, 1158

sorting, 1158

Palindromes, example, 659–660

Paradigm, 815–818, 1221

Parameterization, function objects, 767

Parameterized type, 682–683

Parameters, 1221

functions, 47, 115

list, 115

naming, 273

omitting, 273

templates, 679–681, 687–689

Parametric polymorphism, 682–683

Parsers, 190, 195

**Expression** example, 190, 197–200, 202–203

functions required, 196

grammar rules, 194–195

rules *vs.* tokens, 194

Parsing

expressions, 190–193

grammar, English, 193–194

grammar, programming, 190–193  
tokens, 190–193  
**partial\_sort()**, 1157  
**partial\_sort\_copy()**, 1158  
**partial\_sum()**, 770, 1184  
**partition()**, 1158  
Pascal language, 829–831  
Passing arguments  
    by **const** reference, 276–278, 281–284  
    copies of, 276  
    modified arguments, 278  
    by non-**const** reference, 281–284  
    by reference, 279–284  
    temporary objects, 282  
    unmodified arguments, 277  
    by value, 276, 281–284  
Patterns. *See* Regular expressions  
Performance  
    C++ and C, 1024  
    ideals, 810  
    testing, 1012–1014  
    timing, 1015–1016  
Permutations, 1160–1161  
Petersen, Lawrence, 15  
Pictures. *See* Graphics  
Pivoting, 911–912  
Pixels, 419–420  
**plus()**, 1164  
**Point** example, 445–447  
**pointer**, 1147  
Pointers, 594. *See also* Arrays; Iterators; Memory  
    \* contents of, 594  
    \* pointer to (in declarations), 587, 1099  
     $\square$  subscripting, 594  
    arithmetic, 651–652  
    array. *See* Pointers and arrays  
    casting. *See* Type conversion  
    to class objects, 606–608  
    conversion. *See* Type conversion  
    to current object, **this**, 618–620  
    debugging, 656–659  
    declaration, C-style strings, 1049–1050  
    decrementing, 651–652  
    definition, 587–588, 1221  
    deleted, 657–658  
    explicit type conversion. *See* Type conversion  
    to functions, 1034–1036  
    incrementing, 651–652  
    initializing, 596–598, 657  
    vs. iterators, 1140  
literal (**0**), 1081  
to local variables, 658  
moving around, 651  
to nonexistent elements, 657–658  
null, **0**, 598, 656–657, 1081  
**NULL** macro, 1190  
vs. objects pointed to, 593–594  
out-of-range conditions, 595–596  
palindromes, example, 661–662  
ranges, 595–596  
reading and writing through, 594–596  
semantics, 637  
size, getting, 590–591  
subscripting  $\square$ , 594  
**this**, 676–677  
unknown, 608–610  
**void\***, 608–610  
Pointers and arrays  
    converting array names to, 653–654  
    pointers to array elements, 650–652  
Pointers and inheritance  
    polymorphism, 951–954  
    a problem, 944–948  
    a solution, 947–951  
    user-defined interface class, 947–951  
    **vector** alternative, 947–951  
Pointers and references  
    differences, 610–611  
    inheritance, 612–613  
    list example, 613–622  
    parameters, 611–612  
    **this** pointer, 618–620  
**polar()**, 920, 1183  
Polar coordinates, 920, 1183  
**Polygon** example, 427–428, 458–460, 497  
    vs. **Closed\_polyline**, 458  
    invariants, 460  
Polyline example  
    closed, 456–458  
    marked, 474–476  
    open, 455–456  
    vs. rectangles, 429–431  
Polymorphism  
    ad hoc, 682–683  
    embedded systems, 951–954  
    parametric, 682–683  
    run-time, 504–505  
    templates, 682–683  
Pools, embedded systems, 940–941  
Pop-up menus, 572

**pop\_back()**, 1149  
**pop\_front()**, 1149  
**pop\_heap()**, 1160  
 Portability, 11  
   C++, 1075  
   FLTK, 418, 1204  
 Positioning in files, 393–394  
 Post-conditions, 165–166, 1001–1002, 1221. *See also* Invariants  
 Post-decrement **--**, 1086, 1101  
 Post-increment **++**, 1086, 1101  
 Postfix expressions, 1086  
 Pre-conditions, 163–165, 1001–1002, 1221. *See also* Invariants  
 Pre-decrement **--**, 1087, 1101  
 Pre-increment **++**, 1087, 1101  
 Precedence, in expressions, 1090  
 Precision, numeric, 386–387, 891–895  
 Predicates, 763  
   on class members, 767–768  
   function objects, 1163  
   passing. *See* Function objects  
   searching, 763–764  
 Predictability, 931  
   error handling, 933–934  
   features to avoid, 932  
   memory allocation, 936, 940  
 Preprocessing, 265  
 Preprocessor directives  
   **#define**, macro substitution, 1129  
   **#ifdef**, 1058–1059  
   **#ifndef**, 1059  
   **#include**, including headers, 1129  
 Preprocessor, 1128  
   coding standards, 978–979  
   **prev\_permutation()**, 1161  
 Princeton University, 838  
**print**, character class, 878, 1179  
 Printable characters, identifying, 397  
**printf()** family  
   %, conversion specification, 1187  
   conversion specifications, 1188–1189  
   **gets()**, 1052, 1190–1191  
   output formats, user-defined types, 1189–1190  
   padding, 1188  
   **printf()**, 1050–1051, 1187  
   **scanf()**, 1052–1053, 1190  
   **stderr**, 1189  
   **stdin**, 1189  
   stdio, 1190–1191  
   **stdout**, 1189  
   synchronizing with I/O streams, 1050–1051  
   truncation, 1189  
 Printing  
   error messages, 150–151  
   variable values, 246  
**priority\_queue** container adaptor, 1144  
 Private, 312  
   base classes, 511  
   implementation details, 210, 306–308, 312–313  
   members, 492–493, 505, 511  
   **private:** label, 306, 1037  
 Problem analysis, 175  
   development stages, 176  
   estimating resources, 177  
   problem statement, 176–177  
   prototyping, 178  
   strategy, 176–178  
 Problem statement, 176–177  
 Procedural programming languages, 815–816  
 Programmers. *See also* Programming  
   communication skills, 22  
   computation ideals, 92–94  
   skills requirements, 22–23  
   stereotypes of, 21–22  
   worldwide numbers of, 843  
 Programming, xxiii, 1221. *See also* Computation; Software  
   abstract-first approach, 10  
   analysis stage, 35  
   bottom-up approach, 9  
   C first approach, 9  
   concept-based approach, 6  
   concrete-first approach, 6  
   depth-first approach, 6  
   design stage, 35  
   environments, 52  
   feedback, 36  
   generic, 1219  
   implementation, 36  
   magical approach, 10  
   object-oriented, 10, 1220  
   programming stage, 36  
   software engineering principles first approach, 10  
   stages of, 35–36  
   testing stage, 36  
   top-down approach, 9–10  
   writing a program. *See* Calculator example  
 Programming languages, 818–819, 821, 843  
   Ada, 832–833  
   Algol60, 827–829

- Algol family, 826–829  
assemblers, 820  
auto codes, 820  
BCPL, 838–839  
C, 836–839  
C#, 831  
C++, 839–842  
COBOL, 823–825  
Common Lisp, 825  
Delphi, 831  
Fortran, 821–823  
Lisp, 825–826  
Pascal, 829–831  
Scheme, 825  
Simula, 833–835  
Turbo Pascal, 831  
Programming philosophy, 807, 1221. *See also* C++ and C; Programming ideals; Programming languages  
Programming ideals  
abstraction level, 812–813  
aims, 807–809  
bottom-up approach, 811  
code structure, 810–811  
consistency, 814–815  
correct approaches, 811  
correctness, 810  
data abstraction, 816  
desirable properties, 807–808  
direct expression of ideas, 811–812  
efficiency, 810  
generic programming, 816  
KISS, 815  
maintainability, 810  
minimalism, 814–815  
modularity, 813–814  
multi-paradigm, 818  
object-oriented programming, 815–818  
overview, 808–809  
paradigms, 815–818  
performance, 810  
philosophies, 807–809  
procedural, 815–816  
styles, 815–818  
on-time delivery, 810  
top-down approach, 811  
Programming, history, 818–819. *See also* Programming languages  
BNF (Backus-Naur) Form, 823, 828  
classes, 834  
CODASYL committee, 824  
early languages, 819–821  
first documented bug, 824–825  
first modern stored program, 819–821  
first programming book, 820  
functional programming, 823  
function calls, 820  
inheritance, 834  
K&R, 838  
lint, 836  
object-oriented design, 834  
STL (Standard Template Library), 841  
virtual functions, 834  
Programs, 44, 1221. *See also* Computation; Software audiences for, 46  
compiling. *See* Compilers  
computing values. *See* Expression conforming, 1075  
experimental. *See* Prototyping  
flow, tracing, 72  
implementation defined, 1075  
legal, 1075  
linking, 51  
not-conforming constructs, 1075  
run. *See* Command line; Visual Studio, 52  
starting execution, 46–47, 1075–1076  
stored on a computer, 109  
subdividing, 177–178  
terminating, 208–209, 1075–1076  
text of. *See* Source code  
translation units, 51  
troubleshooting. *See* Debugging  
unspecified constructs, 1075  
valid, 1075  
writing, example. *See* Calculator example  
writing your first, 45–47  
Program organization. *See also* Programming ideals  
abstraction, 92–93  
divide and conquer, 93  
Projects, Visual Studio, 1199–1200  
Promotions, 99, 1091  
Prompting for input, 61  
    >, input prompt, 223  
    calculator example, 179  
    sample code, 223–224  
Proofs, testing, 992  
**protected**, 492–493, 505, 511, 1037  
Prototyping, 178  
Pseudo code, 179, 1221  
Public, 306, 1037  
    base class, 508  
    interface, 210, 496–499

Public, *continued*  
 member, 306  
 public by default, **struct**, 307–308  
**public:** label, 306  
**punct**, punctuation character class, 878, 1179  
**Punct\_stream** example, 401–405  
 Pure virtual functions, 495, 1221  
**push\_back()**  
 growing a **vector**, 119–120  
 queue operations, 1149  
 resizing **vector**, 674–675  
 stack operations, 1149  
 string operations, 1177  
**push\_front()**, 1149  
**push\_heap()**, 1160  
**put()**, 1173  
**putback()**  
 naming convention, 211  
 putting tokens back, 206–207  
 return value, disabling, 211–212  
**putc()**, 1191  
**putchar()**, 1191  
 Putting back input, 206–208

## Q

**qsort()**, 1194–1195  
**<queue>**, 1134  
**queue** container adaptor, 1144  
 Queue operations, 1149

## R

**r** carriage return, character literal, 1079  
**r**, reading file mode, 1186  
**r+**, reading and writing file mode, 1186  
 RAII (Resource Acquisition Is Initialization)  
 definition, 1221  
 exceptions, 700–701, 1125  
 testing, 1004–1005  
 for **vector**, 705–707  
**<random>**, 1134  
 Random numbers, 914–917  
 Random-access iterators, 752, 1142  
 Range  
 definition, 1221  
 errors, 148–150  
 pointers, 595–596  
 regular expressions, 877–878

Range checking  
**at()**, 693–694  
**I**, 650–652, 693–696  
 arrays, 650–652  
 compatibility, 695  
 constraints, 695  
 design considerations, 694–696  
 efficiency, 695  
 exceptions, 693–694  
 macros, 696–697  
 optional checking, 695–696  
 overview, 693–694  
 pointer, 650–652  
**vector**, 693–696  
**range-for**, 119  
**rbegin()**, 1148  
 Re-throwing exceptions, 702, 1126  
**read()**, unformatted input, 1172  
 Readability  
 expressions, 95  
 indenting nested code, 271  
 nested code, 271  
 Reading  
 dividing functions logically, 359–362  
 files. *See* Reading files  
 with iterators, 1140–1141  
 numbers, 214–215  
 potential problems, 358–363  
 separating dialog from function, 362–363  
 a series of values, 356–358  
 a single value, 358–363  
 into **strings**, 851  
 tokens, 185  
 Reading files  
 binary I/O, 391  
 converting representations, 374–376  
 to end of file, 366  
 example, 352–354  
**fstream** type, 350–352  
**ifstream** type, 350–352  
 input loops, 365–367  
**istream** type, 349–354, 391  
 in-memory representation, 368–370  
**ostream** type, 391  
 process steps, 350  
 structured files, 367–376  
 structured values, 370–374  
 symbolic representations, 374–376  
 terminator character, specifying, 366

- real()**, 920, 1183  
Real numbers, 891  
Real part, 920  
Real-time constraints, 931  
Real-time response, 928  
**realloc()**, 1045, 1193  
Recovering from errors, 239–241, 355–358. *See also* Error handling; Exceptions  
**Rectangle** example, 428–431, 460–465, 497  
Recursion  
    definition, 1221  
    infinite, 198, 1220  
    looping, 200  
Recursive function calls, 289  
Red-black trees, 779. *See also* Associative containers; **map**, associative array  
Red margin alerts, 3  
Reference semantics, 637  
References, 1221. *See also* Aliases  
    & in declarations, 276–279  
    to arguments, 277–278  
    circular. *See* Circular reference  
    to last **vector** element, **back()**, 737  
    vs. pointers. *See* Pointers and references  
**<regex>**, 1134, 1175  
**regex**. *See* Regular expressions  
**regex\_error** exception, 1138  
**regex\_match()**, 1177  
    vs. **regex\_search()**, 883  
**regex\_search()**, 1177  
    vs. **regex\_match()**, 883  
**regex** pattern matching, 866–868  
    \$ end of line, 873, 1178  
    () grouping, 867, 873, 876  
    \* zero or more occurrences, 868, 873–874  
    [] character class, 873  
    \ escape character, 866–867, 873  
    \ as literal, 877  
    ^ negation, 873  
    ^ start of line, 873  
    {} count, 867, 873–875  
    | alternative (or), 867–868, 873, 876  
    + one or more occurrences, 873, 874–875  
    . wildcard, 873  
    ? optional occurrence, 867–868, 873, 874–875  
    alternation, 876  
    character classes. *See* **regex** character classes  
        character sets, 877–878  
        definition, 870  
        grouping, 876  
        **matches**, 870  
        pattern matching, 872–873  
        ranges, 877–878  
        **regex** operators, 873, 1177–1179  
        **regex\_match()**, 1177  
        **regex\_search()**, 1177  
        repeating patterns, 874–876  
        searching with, 869–872, 880  
        **smatch**, 870  
        sub-patterns, 867, 870  
**regex** character classes, 877–878  
    **alnum**, 878  
    **alpha**, 878  
    **blank**, 878  
    **cntrl**, 878  
    **d**, 878  
    **\d**, 873  
    **\D**, 873  
    **digit**, 878  
    **graph**, 878  
    **\l**, 873  
    **\L**, 874  
    **lower**, 878  
    **print**, 878  
    **punct**, 878  
    **regex\_match()** vs. **regex\_search()**, 883  
    **s**, 878  
    **\s**, 873  
    **\S**, 874  
    **space**, 878  
    **\u**, 873  
    **\U**, 874  
    **upper**, 878  
    **w**, 878  
    **\w**, 873  
    **\W**, 873  
    **xdigit**, 878  
Regression tests, 993  
Regular expressions, 866–868, 872, 1221.  
    *See also* **regex** pattern matching  
    character classes, 873–874  
    error handling, 878–880  
    grouping, 867, 873, 876  
    uses for, 865  
ZIP code example, 880–885  
Regularity, 380

**reinterpret\_cast**, 609–610, 1095  
 casting unrelated types, 609  
 hardware access, 944  
**Relational operators**, 1088  
**Reliability**, software, 34, 928  
**Remainder and assign %=**, 1090  
**Remainder %** (modulo), 66, 1088  
 correspondence to \* and /, 68  
 floating-point, 201, 230–231  
 integer and floating-point, 66  
**remove()**, 1155  
**remove\_copy()**, 1155  
**remove\_copy\_if()**, 1155  
**rend()**, 1148  
 Repeated words examples, 71–74  
 Repeating patterns, 194  
 Repetition, 1178. *See also* Iteration; **regex**  
**replace()**, 1155  
**replace\_copy()**, 1155  
 Reporting errors  
**Date** example, 317–318  
 debugging, 159  
**error()**, 142–143  
 run-time, 145–146  
 syntax errors, 137–138  
 Representation, 305, 671–673  
 Requirements, 1221. *See also* Invariants; Post-conditions; Pre-conditions  
 for functions, 153  
**reserve()**, 673–674, 691, 747, 1151  
 Reserved names, 75–76. *See also* Keywords  
**resetiosflags()** manipulator, 1174  
**resize()**, 674, 1151  
 Resource, 1221  
 leaks, 931, 934  
 limitations, 928  
 management. *See* Resource management  
 testing, 1001–1002  
**vector** example, 697–698  
 Resource Acquisition Is Initialization (RAII), 1221  
 exceptions, 700–701, 1125  
 testing, 1004–1005  
 for **vector**, 705–707  
 Resource management, 697–702. *See also* **vector** example  
 basic guarantee, 702  
 error handling, 702  
 guarantees, 701–702  
**make\_vec()**, 702  
 no-throw guarantee, 702  
 problems, 698–700  
 RAII, 700–701, 705–707  
 resources, examples, 697–698  
 strong guarantee, 702  
 testing, 1004–1005  
 Results, 91. *See also* Return values  
**return** and move, 704–705  
**return** statement, 272–273  
 Return types, functions, 47, 272–273  
 Return values, 113–115  
 functions, 1103  
 no return value, **void**, 212  
 omitting, 115  
 returning, 272–273  
**reverse()**, 1155  
**reverse\_copy()**, 1155  
**reverse\_iterator**, 1147  
 Revision history, 237–238  
 Rho, 920  
 Richards, Martin, 838  
**right** manipulator, 1174  
 Ritchie, Dennis, 836, 837, 842, 1022–1023, 1032  
 Robot-assisted surgery, 30  
**rotate()**, 1155  
**rotate\_copy()**, 1155  
 Rounding, 386, 1221. *See also* Truncation  
 errors, 891  
 floating-point values, 386  
 Rows, matrices, 900–901, 906  
 Rules, for programming. *See* Ideals  
 Rules, grammatical, 194–195  
 Run-time dispatch, 504–505. *See also* Virtual functions  
 Run-time errors. *See* Errors, run-time  
 Run-time polymorphism, 504–505  
**runtime\_error**, 142, 151, 153  
 rvalue reference, 639  
 Rvalues, 94–95, 1090

## S

**s**, character class, 878, 1179  
**\S**, “not space,” **regex**, 874  
**\s**, “space,” **regex**, 873  
 Safe conversions, 79–80  
 Safety, type. *See* Type, safety  
 Scaffolding, cleaning up, 234–235  
**scale\_and\_add()** example, 904  
**scale\_and\_multiply()** example, 912  
 Scaling data, 542–543

**scanf()**, 1052, 1190  
Scenarios. *See* Use cases  
Scheme language, 825  
**scientific** format, 387  
**scientific** manipulator, 385, 1174  
Scope, 266–267, 1082–1083, 1221  
    class, 267, 1082  
    enumerators, 320–321  
    global, 267, 270, 1082  
    going out of, 268–269  
    kinds of, 267  
    local, 267, 1083  
    namespace, 267, 271, 1082  
    resolution `::`, 295–296, 1086  
    statement, 267, 1083  
Scope and nesting  
    blocks within functions, 271  
    classes within classes, 270  
    classes within functions, 270  
    functions within classes, 270  
    functions within functions, 271  
    indenting nested code, 271  
    local classes, 270  
    local functions, 270  
    member classes, 270  
    member functions, 270  
    nested blocks, 271  
    nested classes, 270  
    nested functions, 270  
Scope and object lifetime, 1085–1086  
    free-store objects, 1085  
    local (automatic) objects, 1085  
    namespace objects, 1085  
    static class members, 1085  
    temporary objects, 1085  
Scope and storage class, 1083–1084  
    automatic storage, 1083–1084  
    free store (heap), 1084  
    static storage, 1084  
Screens. *See also* GUIs (graphical user interfaces)  
    data graph layout, 541–542  
    drawing on, 423–424  
    labeling, 425  
**search()**, 795–796, 1153  
Searching. *See also* Finding; Matching; **find\_if()**; **find()**  
    algorithms for, 1157–1159  
    binary searches, 779, 795–796  
    in C, 1194–1195  
    for characters, 740  
    (key,value) pairs, by key. *See* Associative containers  
    for links, 615–617  
    **map** elements. *See* **unordered\_map**  
    predicates, 763  
    with regular expressions, 869–872, 880–885, 1177–1179  
**search\_n()**, 1153  
Self reference. *See* **this** pointer  
Self assignment, 676–677  
Self-checking, error handling, 934  
Separators, nonstandard, 398–405  
Sequence containers, 1144  
Sequences, 720, 1221  
    algorithms. *See* Algorithms, STL  
    differences between adjacent elements, 770  
    empty, 729  
    example, 723–724  
    half open, 721  
Sequencing rules, 195  
Server farms, 31–32  
**set**, 776, 787–789  
    iterators, 1144  
    *vs.* **map**, 788  
    subscripting, 788  
**set()**, 605–606  
**<set>**, 776, 1134  
Set algorithms, 1159–1160  
**set\_difference()**, 1160  
**set\_intersection()**, 1159  
**set\_symmetric\_difference()**, 1160  
**set\_union()**, 1159  
**setbase()** manipulator, 1174  
**setfill()** manipulator, 1174  
**setiosflags()** manipulator, 1174  
**setprecision()** manipulator, 386–387, 1174  
**setw()** manipulator, 1174  
Shallow copies, 636  
**Shape** example, 493–494  
    abstract classes, 495–496  
    access control, 496–499  
    attaching to **Window**, 545–546  
    as base class, 445, 495–496  
    **clone()**, 504  
    copying objects, 503–504  
    **draw()**, 500–502  
    **draw\_lines()**, 500–502  
    fill color, 500  
    implementation inheritance, 513–514  
    interface inheritance, 513–514

**Shape** example, *continued*  
 line visibility, 500  
**move()**, 502  
 mutability, 503–504  
**number\_of\_points()**, 449  
 object layout, 506–507  
 object-oriented programming, 513–514  
**point()**, 449  
 slicing shapes, 504  
 virtual function calls, 501, 506–507  
 Shift operators, 1088  
 Shipping, computer use, 26–28  
**short**, 955, 1099  
 Shorthand notation, regular expressions, 1179  
**showbase**, manipulator, 383, 1173  
**showpoint**, manipulator, 1173  
**showpos**, manipulator, 1173  
 Shuffle algorithm, 1155–1156  
 Signed and unsigned integers, 961–965  
**signed** type, 1099  
**Simple\_window**, 422–424, 443  
 Simplicity ideal, 92–94  
 Simula language, 833–835  
**sin()**, sine, 917, 1182  
 Singly-linked lists, 613, 725  
**sinh()**, hyperbolic sine, 918, 1182  
 Size  
   bit strings, 955–956  
   containers, 1150–1151  
   getting, **sizeof()**, 590–591  
   of numbers, 891–895  
   **vectors**, getting, 119–120  
**size()**  
   container capacity, 1150  
   number of elements, 120, 851  
   string length, 851, 1176  
   **vectors**, 120, 122–123  
**sizeof()**, 590–591, 1094  
   object size, 1087  
   value size, 892  
**size\_type**, 730, 1147  
**skipws**, 1174  
**slice()**, 901–902, 905  
 Slicing  
   matrices, 901–902, 905  
   objects, 504  
 Smallest integer, finding, 917  
**smatch**, 870  
 Soft real-time, 931  
 Software, 19, 1222. *See also* Programming; Programs  
   affordability, 34  
   correctness, 34  
   ideals, 34–37  
   maintainability, 35  
   reliability, 34  
   troubleshooting. *See* Debugging  
   useful design, 34  
   uses for, 19–33  
 Software layers, GUIs, 557  
**sort()**, 758, 794–796, 1157  
**sort\_heap()**, 1160  
 Sorting  
   algorithms for, 1157–1159  
   in C, **qsort()**, 1194  
   **sort()**, 758, 794–796, 1157  
 Source code  
   definition, 48, 1222  
   entering, 1200  
 Source files, 48, 1222  
   adding to projects, 1200  
**space**, 878, 1179  
 Space exploration, computer use, 33  
 Special characters, 1079–1080  
   regular expressions, 1178  
 Specialization, 681, 1123  
 Specifications  
   definition, 1221  
   source of errors, 136  
 Speed of light, 96  
**sprintf()**, 1187  
**sqrt()**, square root, 917, 1181  
 Square of **abs()**, norm, 919  
**<sstream>**, 1134  
**stable\_partition()**, 1158  
**stable\_sort()**, 1157  
**<stack>**, 1134  
**stack** container adaptor, 1144  
 Stack of activation records, 287  
 Stack storage, 591–592  
 Stacks  
   container operations, 1149  
   embedded systems, 935–936, 940, 942–943  
   growth, 287–290  
   unwinding, 1126  
 Stages of programming, 35–36  
 Standard  
   conformance, 836, 974, 1075  
   ISO, 1075, 1222

manipulators. *See* Manipulators  
mathematical functions, 917–918  
Standard library. *See also* C standard library; STL  
(Standard Template Library)  
algorithms. *See* Algorithms  
**complex**. *See* **complex**  
containers. *See* Containers  
C-style I/O. *See* **printf()** family  
C-style strings. *See* C-style strings  
date and time, 1193–1194  
function objects. *See* Function objects  
I/O streams. *See* Input; Input/output;  
    Output  
iterators. *See* Iterators  
mathematical functions. *See* Mathematical  
    functions (standard)  
numerical algorithms. *See* Algorithms,  
    numerical; Numerics  
**string**. *See* **string**  
time, 1015–1016, 1193  
**valarray**. *See* **valarray**  
Standard library header files, 1133–1136  
algorithms, 1133–1134  
containers, 1133–1134  
C standard libraries, 1135–1136  
I/O streams, 1134  
iterators, 1133–1134  
numerics, 1134–1135  
string manipulation, 1134  
utility and language support, 1135  
Standard library I/O streams, 1168–1169. *See also*  
    I/O streams  
Standard library string manipulation  
    character classification, 1175–1176  
    containers. *See* **map**, associative array; **set**;  
        **unordered\_map**; **vector**  
    input/output. *See* I/O streams  
    regular expressions. *See* **regex**  
    string manipulation. *See* **string**  
Stanford University, 826  
Starting programs, 1075–1076. *See also* **main()**  
State, 90–91, 1222  
    I/O stream, 1171  
    of objects, 305  
    source of errors, 136  
    testing, 1001  
    validity checking, 313  
    valid state, 313  
Statement scope, 267, 1083  
Statements, 47  
    grammar, 1096–1097  
    named sequence of. *See* Function  
    terminator ; (semicolon), 50, 100  
Static storage, 591–592, 1084  
    class members, lifetime, 1085  
    embedded systems, 935–936, 944  
    **static**, 1084  
    **static const**, 326. *See also* **const**  
    **static** local variables, order of initialization, 294  
**std** namespace, 296–297, 1136  
**stderr**, 1189  
**<stdexcept>**, 1135  
**stdin**, 1050, 1189. *See also* stdio  
stdio, standard C I/O, 1050, 1190–1191  
    **EOF** macro, 1053–1054  
    **errno**, error indicator, 918–919  
    **fclose()**, 1053–1054  
    **FILE**, 1053–1054  
    **fopen()**, 1053–1054  
    **getchar()**, 1052–1053, 1191  
    **gets()**, 1052, 1190–1191  
    input, 1052–1053  
    output, 1050–1051  
    **printf()**, 1050–1051, 1188–1191  
    **scanf()**, 1052, 1190  
    **stderr**, **cerr** equivalent, 1189  
    **stdin**, **cin** equivalent, 1050, 1189  
    **stdout**, 1050, 1189. *See also* stdio  
    **stdout**, **cout** equivalent, 1050, 1189  
**std\_lib\_facilities.h** header file, 1199–1200  
**stdout**, 1050, 1189. *See also* stdio  
Stepanov, Alexander, 720, 722, 841  
Stepping through code, 162  
Stereotypes of programmers, 21–22  
STL (Standard Template Library), 717, 1149–  
    1168 (large range, not sure this is correct). *See*  
    *also* C standard library; Standard library  
algorithms. *See* STL algorithms  
containers. *See* STL containers  
function objects. *See* STL function objects  
history of, 841  
ideals, 717–720  
iterators. *See* STL iterators  
namespace, **std**, 1136  
STL algorithms, 1152–1162  
    *See* Algorithms, STL.  
alternatives to, 1195  
built-in arrays, 747–749

STL algorithms, *continued*  
 computation *vs.* data, 717–720  
 heap, 1160  
**max()**, 1161  
**min()**, 1161  
 modifying sequence, 1154–1156  
 mutating sequence, 1154–1156  
 nonmodifying sequence, 1153–1154  
 permutations, 1160–1161  
 searching, 1157–1159  
 set, 1159–1160  
 shuffle, 1155–1156  
 sorting, 1157–1159  
 utility, 1157  
 value comparisons, 1161–1162

STL containers, 749–751, 1144–1152  
 almost, 751, 1145  
 assignments, 1148  
 associative, 1144, 1151–1152  
 capacity, 1150–1151  
 comparing, 1151  
 constructors, 1148  
 container adaptors, 1144  
 copying, 1151  
 destructors, 1148  
 element access, 1149  
 information sources about, 750  
 iterator categories for, 752, 1143–1145,  
   1148  
 list operations, 1150  
 member types, 1147  
 operations overview, 1146–1147  
 queue operations, 1149  
 sequence, 1144  
 size, 1150–1151  
 stack operations, 1149  
 swapping, 1151

STL function objects, 1163  
 adaptors, 1164  
 arithmetic operations, 1164  
 inserters, 1162–1163  
 predicates, 767–768, 1163

STL iterators, 1139–1140  
 basic operations, 721  
 categories, 1142–1143  
 definition, 721, 1139  
 description, 721–722  
 empty lists, 729  
 example, 737–741

operations, 1141–1142  
*vs.* pointers, 1140  
 sequence of elements, 1140–1141

Storage class, 1083–1084  
 automatic storage, 1083–1084  
 free store (heap), 1084  
 static storage, 1084

Storing data. *See* Containers  
**str()**, **string** extractor, 395  
**strcat()**, 1047, 1191  
**strchr()**, 1048, 1192  
**strcmp()**, 1047, 1192  
**strcpy()**, 1047, 1049, 1192

Stream  
 buffers, 1169  
 iterators, 790–793  
 modes, 1170  
 states, 355  
 types, 1170  
**streambuf**, 406, 1169  
**<streambuf>**, 1134  
**<string>**, 1134, 1172  
**string**, 66, 851, 1222. *See also* Text  
 [] subscripting, 851  
 + concatenation, 68–69, 851, 1176  
 += append, 851  
 < lexicographical comparison, 851  
 == equal, 851  
 = assign, 851  
 >> input, 851  
 << output, 851  
 almost container, 1145  
**append()**, 851  
**basic\_string**, 852  
 C++ to C-style conversion, 851  
**c\_str()**, C++ to C-style conversion, 851  
**erase()**, removing characters, 851  
 exceptions, 1138  
**find()**, 851  
**from\_string()**, 853–854  
**getline()**, 851  
 input terminator (whitespace), 65  
**Insert()**, adding characters, 851  
**length()**, number of characters, 851  
**lexical\_cast** example, 855  
 literals, debugging, 161  
 operations, 851, 1176–1177  
 operators, 66–67, 68  
 palindromes, example, 659–660

- pattern matching. *See* Regular expressions  
properties, 741–742  
size, 78  
**size()**, number of characters, 851  
standard library, 852  
**stringstream**, 852–854  
**string** to value conversion, 853–854  
subscripting **[ ]**, 851  
**to\_string()** example, 852–854  
values to string conversion, 852  
vs. **vector**, 745  
whitespace, 854  
String literal, 62, 1080  
**stringstream**, 395, 852–854, 1170  
**strlen()**, 1046, 1191  
**strncat()**, 1047, 1192  
**strcmp()**, 1047, 1192  
**strcpy()**, 1047, 1192  
Strong guarantee, 702  
Stroustrup, Bjarne  
    advisor, 820  
    Bell Labs colleagues, 836–839, 1023  
    biography, 13–14  
    education on invariants, 828  
    inventor of C++, 839–842  
    Kristen Nygaard, 834  
**strpbrk()**, 1192  
**strchr()**, 1192  
**strstr()**, 1192  
**strtod()**, 1192  
**strtol()**, 1192  
**strtoul()**, 1192  
**struct**, 307–308. *See also* Data  
**struct** tag namespace, 1036–1037  
Structure  
    of data. *See* Data  
    of programs, 215–216  
Structured files, 367–376  
Style, definition, 1222  
Sub-patterns, 867, 870  
Subclasses, 504. *See also* Derived classes  
Subdividing programs, 177–178  
Subscripting, 118  
    **0** Fortran style, 899  
    **[ ]** C Style, 694, 899  
    arrays, 649, 899  
    **at()**, checked subscripting, 694, 1149  
    **Matrix** example, 899–901, 905  
    pointers, 1101
- string**, 851, 1176  
**vector**, 594, 607–608, 646–647  
Substrings, 863  
Subtraction **-** (minus)  
**complex**, 919, 1183  
definition, 1088  
integers, 1101  
iterators, 1141–1142  
pointers, 1101  
Subtype, definition, 1222  
Summing values. *See* **accumulate()**  
Superclasses, 504, 1222. *See also* Base classes  
**swap()**, 281, 1151, 1157  
Swapping  
    columns, 906  
    containers, 1151  
    ranges, 1157  
    rows, 906, 912  
**swap\_ranges()**, 1157  
**switch**-statements  
    **break**, **case** termination, 106–108  
    **case** labels, 106–108  
    most common error, 108  
    vs. **string**-based selection, 106  
Symbol tables, 247  
Symbolic constants. *See also* Enumerations  
    cleaning up, 232–234  
    defining, with **static const**, 326  
Symbolic names, tokens, 233  
Symbolic representations, reading, 374–376  
Syntax analyzers, 190  
Syntax checking, 48–50  
Syntax errors  
    examples, 48–50  
    overview, 137–138  
    reporting, 137–138  
Syntax macros, 1058  
**system()**, 1194  
**system\_clock**, 1016, 1185  
System, definition, 1222  
System tests, 1009–1011

## T

- \t** tab character, 109, 1079  
**tan()**, tangent, 917, 1182  
**tanh()**, hyperbolic tangent, 917, 1182  
TEA (Tiny Encryption Algorithm), 820, 969–974  
Technical University of Copenhagen, 828

- Telecommunications, 28–29
- Temperature data, example, 120–123
- template**, 1038
- Template, 678–679, 1121–1122, 1222
  - arguments, 1122–1123
  - class, 681–683. *See also* Class template
  - compiling, 684
  - containers, 686–687
  - error diagnostics, 683
  - function, 682–690. *See also* Function template
  - generic programming, 682–683
  - inheritance, 686–687
  - instantiation, 681, 1123–1124
  - integer parameters, 687–689
  - member types, 1124
  - parameters, 679–681, 687–689
  - parametric polymorphism, 682–683
  - specialization, 1123
- typename**, 1124
- type parameters, 679–681
- weaknesses, 683
- Template-style casts, 1040
- Temporary objects, 282, 1085
- Terminals, in grammars. *See* Tokens
- Termination
  - abort()** a program, 1194
  - on exceptions, 142
  - exit()** a program, 1194
  - input, 61–62, 179
  - normal program termination, 1075–1076
  - for **string** input, 65
  - zero, for C-style strings, 654–655
- Terminator character, specifying, 366
- Testing, 992–993, 1222. *See also* Debugging
  - algorithms, 1001–1008
  - for bad input, 103
  - black box, 992–993
  - branching, 1006–1008
  - bug reports, retention period, 993
  - calculator example, 225
  - code coverage, 1008
  - debugging, 1012
  - dependencies, 1002–1003
  - designing for, 1011–1012
  - faulty assumptions, 1009–1011
  - files, after opening, 352
  - FLTK, 1206
  - inputs, 1001
  - loops, 1005–1006
  - non-algorithms, 1001–1008
  - outputs, 1001
  - performance, 1012–1014
  - pre- and post-conditions, 1001–1002
  - proofs, 992
  - RAII, 1004–1005
  - regression tests, 993
  - resource management, 1004–1005
  - resources, 1001–1002
  - stage of programming, 36
  - state, 1001
  - system tests, 1009–1011
  - test cases, definition, 166
  - test harness, 997–999
  - timing, 1015–1016
  - white box, 992–993
- Testing units
  - formal specification, 994–995
  - random sequences, 999–1001
  - strategy for, 995–997
  - systematic testing, 994–995
  - test harness, 997–999
- Text
  - character strings. *See* C-style strings; **string**
  - email example, 856–861, 864–865
  - extracting text from files, 855–861, 864–865
  - finding patterns, 864–865, 869–872
  - in graphics. *See* Text
  - implementation details, 861–864
  - input/output, GUIs, 563–564
  - maps. *See* map
  - storage, 591–592
  - substrings, 863
  - vector** example, 123–125
  - words frequency example, 777–779
- Text** example, 431–433, 467–470
- Text editor example, 737–741
- Theta, 920
- this** pointer, 618–620, 676–677
- Thompson, Ken, 836–838
- Three-way comparison, 1046
- Throwing exceptions, 147, 1125
  - I/O stream, 1171
  - re-throwing, 702
  - standard library, 1138–1139
  - throw, 147, 1090, 1125–1126
  - vector**, 697–698
- Time
  - date and time, 1193–1194
  - measuring, 1015–1016
- Timekeeping, computer use, 26

- time\_point**, 1016  
**time\_t**, 1193  
Tiny Encryption Algorithm (TEA), 820, 969–974  
**tm**, 1193  
**Token** example, 183–184  
**Token\_stream** example, 206–214  
**tolower()**, 398, 1176  
Top-down approach, 9–10, 811  
**to\_string()** example, 852–854  
**toupper()**, 398, 1176  
Tracing code execution, 162–163  
Trade-off, definition, 1222  
**transform()**, 1154  
Transient errors, handling, 934  
Translation units, 51, 139–140  
Transparency, 451, 463  
Tree structure, **map** container, 779–782  
**true**, 1037, 1038  
**trunc** mode, 389, 1170  
Truncation, 82, 1222  
    C-style I/O, 1189  
    exceptions, 153  
    floating-point numbers, 893  
**try-catch**, 146–153, 693–694, 1037  
Turbo Pascal language, 831  
Two-dimensional matrices, 904–906  
Two’s complement, 961  
Type, 60, 77, 1222  
    aliases, 730  
    built-in. *See* Built-in types  
    checking, C++ and C, 1032–1033  
    generators, 681  
    graphics classes, 488–490  
    mismatch errors, 138–139  
    mixing in expressions, 99  
    naming. *See* Namespaces  
    objects, 77–78  
    operations, 305  
    organizing. *See* Namespaces  
    parameterized, 682–683. *See also* Template  
    as parameters. *See* Template  
    pointers. *See* Pointer  
    promotion, 99  
    representation of object, 308–309, 506–507  
    safety, 78–79, 82  
    subtype, 1222  
    supertype, 1222  
    truncation, 82  
    user-defined. *See* UDTs (user-defined types)  
    uses for, 304  
values, 77  
variables. *See* Variables  
Type conversion  
    casting, 609–610  
    **const\_cast**, casting away **const**, 609–610  
    exceptions, 153  
    **explicit**, 609  
    in expressions, 99–100  
    function arguments, 284–285  
    implicit, 642–643  
    **int** to pointer, 590  
    operators, 1095  
    pointers, 590, 609–610  
    **reinterpret\_cast**, 609  
    safety, 79–83  
    **static\_cast**, 609  
    **string** to value, 853–854  
    truncation, 82  
    value to **string**, 852  
Type conversion, implicit, 642–643  
**bool**, 1092  
compiler warnings, 1091  
floating-point and integral, 1091–1092  
integral promotion, 1091  
pointer and reference, 1092  
preserving values, 1091  
promotions, 1091  
user-defined, 1091  
usual arithmetic, 1092  
Type safety, 78–79  
    implicit conversions, 80–83  
    narrowing conversions, 80–83  
    pointers, 596–598, 656–659  
    range error, 148–150, 595–596  
    safe conversions, 79–80  
    unsafe conversions, 80–83  
**typeid**, 730  
**typeid**, 1037, 1087, 1138  
**<typeinfo>**, 1135  
**typename**, 1037, 1124

## U

- u/U** suffix, 1077  
**\u**, “not uppercase,” **regex**, 874  
**\u**, “uppercase character,” **regex**, 873, 1179  
UDTs (user-defined types). *See* Class;  
    Enumerations  
Unary expressions, 1087  
“Uncaught exception” error, 153

- Unchecked conversions, 943–944  
 “Undeclared identifier” error, 258  
 Undefined order of evaluation, 263  
**`unget()`**, 355–358  
**`ungetc()`**, 1191  
 Uninitialized variables, 327–330, 1222  
**`uninitialized_copy()`**, 1157  
**`uninitialized_fill()`**, 1157  
**`union`**, 1121  
**`unique()`**, 1155  
**`unique_copy()`**, 758, 789, 792–793, 1155  
**`unique_ptr`**, 703–704  
 Unit tests  
   formal specification, 994–995  
   random sequences, 999–1001  
   strategy for, 995–997  
   systematic testing, 994–995  
   test harness, 997–999  
 Universal and uniform initialization, 83  
 Unnamed objects, 465–467  
**`<unordered_map>`**, 776, 1134  
**`unordered_map`**, 776. *See also map*, associative  
   array  
   finding elements, 785–787  
   hashing, 785  
   hash tables, 785  
   hash values, 785  
   iterators, 1144  
**`unordered_multimap`**, 776, 1144  
**`unordered_multiset`**, 776, 1144  
**`<unordered_set>`**, 776, 1134  
**`unordered_set`**, 776, 1144  
 Unsafe conversions, 80–83  
**`unsetf()`**, 384  
 Unsigned and signed, 961–965  
**`unsigned`** type, 1099  
 Unspecified constructs, 1075  
**`upper`**, character class, 878, 1179  
**`upper_bound()`**, 796, 1152, 1158  
 Uppercase. *See* Case (of characters)  
**`uppercase`**, 1174  
 U.S. Department of Defense, 832  
 U.S. Navy, 824  
 Use cases, 179, 1222  
 User-defined conversions, 1091  
 User-defined operators, 1091  
 User-defined types (UDTs), 304. *See also Class*;  
   Enumerations  
   exceptions, 1126  
   operator overloading, 1107  
 operators, 1107  
   standard library types, 304  
 User interfaces  
   console input/output, 552  
   graphical. *See GUIs (graphical user interfaces)*  
   web browser, 552–553  
**`using`** declarations, 296–297  
**`using`** directives, 296–297, 1127  
 Usual arithmetic conversions, 1092  
 Utilities, STL  
   function objects, 1163–1164  
   inserters, 1162–1163  
**`make_pair()`**, 1165–1166  
**`pair`**, 1165–1166  
**`<utility>`**, 1134, 1165–1166  
 Utility algorithms, 1157  
 Utility and language support, header files,  
   1135

## V

- v** vertical tab, character literal, 1079  
**`valarray`**, 1145, 1183  
**`<valarray>`**, 1135  
 Valid pointer, 598  
 Valid programs, 1075  
 Valid state, 313  
 Validity checking, 313  
   constructors, 313  
   enumerations, 320  
   invariants, 313  
   rules for, 313  
 Value semantics, 637  
**`value_comp()`**, 1152  
 Values, 77–78, 1222  
   symbolic constants for. *See* Enumerations  
   and variables, 62, 73–74, 243  
**`value_type`**, 1147  
 Variables, 62–63, 1083  
**`++`** increment, 73–74  
**`=`** assignment, 69–73  
   changing values, 73–74  
   composite assignment operators, 73–74  
   constructing, 291–292  
   declarations, 260, 262–263  
   going out of scope, 291  
   incrementing **`++`**, 73–74  
   initialization, 69–73  
   input, 60  
   naming, 74–77

- type of, 66–67  
uninitialized, class interfaces, 327–330  
value of, 73–74
- <vector>**, 1134
- vector** example, 584–587, 629–636, 668–679
- subscripting, 646, 693–697
  - = assignment, 675–677
  - . (dot) access, 607–608
  - allocators, 691
  - changing size, 668–679
  - at()**, checked subscripting, 694
  - copying, 631–636
  - destructor, 601–605
  - element type as parameter, 679–681
  - erase()** (removing elements), 745–747
  - exceptions, 693–694, 705–707
  - explicit** constructors, 642–643
  - inheritance, 686–687
  - insert()** (adding elements), 745–747
  - overloading on **const**, 647–648
  - push\_back()**, 674–675, 692
  - representation, 671–673
  - reserve()**, 673, 691, 704–705
  - resize()**, 674, 692
  - subscripting, 594, 607–608, 646–647
- vector**, standard library, 1146–1151
- subscripting, 1149
  - = assignment, 1148
  - == equality, 1151
  - < less than, 1151
  - assign()**, 1148
  - back()**, reference to last element, 1149
  - begin()**, iterator to first element, 1148
  - capacity()**, 1151
  - at()**, checked subscripting, 1149
  - const\_iterator**, 1147
  - constructors, 1148
  - destructor, 1148
  - difference\_type**, 1147
  - end()**, one beyond last element, 1148
  - erase()**, removing elements, 1150
  - front()**, reference to first element, 1149
  - insert()**, adding elements, 1150
  - iterator**, 1147
  - member functions, lists of, 1147–1151
  - member types, list of, 1147
  - push\_back()**, add element at end, 1149
  - size()**, number of elements, 1151
  - size\_type**, 1147
  - value\_type**, 1147
- vector** of references, simulating, 1212–1213
- Vector\_ref** example, 444, 1212–1213
- vector\_size()**, 119
- virtual**, 1037
- Virtual destructors, 604–605. *See also* Destructors
- Virtual functions, 501, 506–507
- declaring, 508
  - definition, 501, 1222
  - history of, 834
  - object layout, 506–507
  - overriding, 508–511
  - pure, 512–513
- Shape** example, 501, 506–507
- vptr**, 506–507
- vtbl**, 506
- Visibility. *See also* Scope; Transparency
- menus, 573–574
  - of names, 266–272, 294–297
  - widgets, 562
- Visual Studio
- FLTK (Fast Light Toolkit), 1205–1206
  - installing, 1198
  - running programs, 1199–1200
- void**, 115
- function results, 115, 273, 275
  - pointer to, 608–610
- putback()**, 212
- void\***, 608–610, 1041–1042, 1099
- vptr**, virtual function pointer, 506–507
- vtbl**, virtual function table, 506
- W**
- w**, writing file mode, 878, 1179, 1186
- w+**, writing and reading file mode, 1186
- \W**, “not word character,” **regex**, 874, 1179
- \w**, “word character,” **regex**, 873, 1179
- wait()**, 559–560, 569–570
- Wait loops, 559–560
- wait\_for\_button()** example, 559–560
- Waiting for user action, 559–560, 569–570
- wchar\_t**, 1038
- Web browser, as user interface, 552–553
- Wheeler, David, 109, 820, 954, 969
- while**-statements, 109–111
  - vs. **for**, 122
- White-box testing, 992–993
- Whitespace
- formatting, 397, 398–405
  - identifying, 397

- Whitespace  
in input, 64  
**string**, 854
- Widget** example, 561–563  
**Button**, 422–424, 553–561  
control inversion, 569–570  
debugging, 576–577  
**hide()**, 562  
implementation, 1209–1210  
**In\_box()**, 563–564  
line drawing example, 565–569  
**Menu**, 564–565, 570–575  
**move()**, 562  
**Out\_box()**, 563–564  
**put\_on\_top()**, 1211  
**show()**, 562  
technical example, 1213–12116  
text input/output, 563–564  
visibility, 562
- Wild cards, regular expressions, 1178
- Wilkes, Maurice, 820
- Window** example, 420, 443  
canvas, 420  
creating, 422–424, 554–556  
disappearing, 576  
drawing area, 420  
implementation, 1210–1212
- line drawing example, 565–569  
**put\_on\_top()**, 1211
- Window.h** example, 421–422
- Wirth, Niklaus, 830–831
- Word frequency, example, 777
- Words (of memory), 1222
- write()**, unformatted output, 1173
- Writing files, 350. *See also* File I/O  
appending to, 389  
binary I/O, 391  
example, 352–354  
**fstream** type, 350–352  
**ofstream** type, 351–352  
**ostream** type, 349–354, 391
- ws** manipulator, 1174
- X**
- xdigit**, 878, 1179
- \xhhh**, hexadecimal character literal, 1080
- xor**, synonym for **^**, 1038
- xor\_eq**, synonym for **^=**, 1038
- Z**
- zero-terminated array, 1045. *See also* C-style strings
- ZIP code example, 880–885

# Photo Citations and Credits

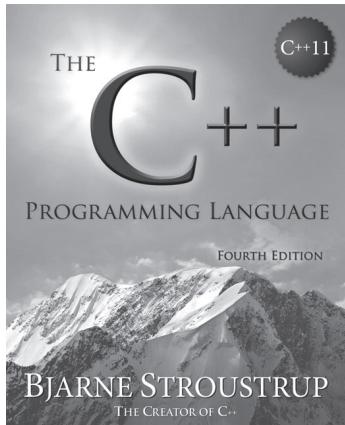
- Page 14. Photo of Bjarne Stroustrup, 2005. Source: Bjarne Stroustrup.
- Page 15. Photo of Lawrence “Pete” Petersen, 2006. Source: Dept. of Computer Science, Texas A&M University.
- Page 26. Photo of digital watch from Casio. Source: [www.casio.com](http://www.casio.com).
- Page 26. Photo of analog watch from Casio. Source: [www.casio.com](http://www.casio.com).
- Page 26. MAN marine diesel engine 12K98ME; MAN Burgmeister & Waine. Source: MAN Diesel A/S, Copenhagen, Denmark.
- Page 26. Emma Maersk; the world’s largest container ship; home port Århus, Denmark. Source: Getty Images.
- Page 28. Digital telephone switchboard. Source: Alamy Images.
- Page 28. Sony-Ericsson W-920 cell phone with music system, cell phone, and web connectivity. Source: [www.sonyericsson.com](http://www.sonyericsson.com).
- Page 29. Trading floor of the New York Stock Exchange in Wall Street. Source: Alamy Images.
- Page 29. A representation of parts of the internet backbone by Stephen G. Eick. Source: S. G. Eick.
- Page 30. CAT scanner. Source: Alamy Images.
- Page 30. Computer-aided surgery. Source: Da Vinci Surgical Systems, [www.intuitivesurgical.com](http://www.intuitivesurgical.com).
- Page 31. Ordinary computer setup (the left-hand screen is connected to a Unix desktop box, the right-hand screen is a Windows laptop). Source: Bjarne Stroustrup.
- Page 31. Computer rack from a server farm. Source: Istockphoto.
- Page 33. View from a Mars rover. Source: NASA, [www.nasa.gov](http://www.nasa.gov).
- Page 820. The EDSAC team 1949. Maurice Wilkes center, David Wheeler without a tie. Source: The Cambridge University Computer Laboratory.
- Page 820. David Wheeler lecturing circa 1974. Source: University of Cambridge Computer Laboratory.

- Page 822. John Backus 1996. Copyright: Louis Fabian Bachrach. For a collection of photographs of computer pioneers, see Christopher Morgan: *Wizards and their wonders: portraits in computing*. ACM Press. 1997. ISBN 0-89791-960-2.
- Page 824. Grace Murray Hopper. Source: Computer History Museum.
- Page 825. Grace Murray Hopper's bug. Source: Computer History Museum.
- Page 826. John C. McCarthy, 1967, at Stanford. Source: Stanford University.
- Page 826. John C. McCarthy, 1996. Copyright: Louis Fabian Bachrach.
- Page 827. Peter Naur photographed by Brian Randell in Munich in 1968 when they together edited the report that launched the field of Software Engineering. Reproduced by permission from Brian Randell.
- Page 827. Peter Naur, from oil painting by Duo Duo Zhuang 1995. Reproduced by permission from Erik Frøkjær.
- Page 828. Edsger Dijkstra. Source: Wikimedia Commons.
- Page 830. Niklaus Wirth. Source: N. Wirth.
- Page 830. Niklaus Wirth. Source: N. Wirth.
- Page 832. Jean Ichbiah. Source: Ada Information Clearinghouse.
- Page 832. Lady Lovelace, 1838. Vintage print. Source: Ada Information Clearinghouse.
- Page 834. Kristen Nygaard and Ole-Johan Dahl, circa 1968. Source: University of Oslo.
- Page 835. Kristen Nygaard, circa 1996. Source: University of Oslo.
- Page 835. Ole-Johan Dahl, 2002. Source: University of Oslo.
- Page 836. Dennis M. Ritchie and Ken Thompson, approx. 1978. Copyright: AT&T Bell Labs.
- Page 836. Dennis M. Ritchie, 1996. Copyright: Louis Fabian Bachrach.
- Page 837. Doug McIlroy, circa 1990. Source: Gerard Holzmann.
- Page 837. Brian W. Kernighan, circa 2004. Source: Brian Kernighan.
- Page 840. Bjarne Stroustrup, 1996. Source: Bjarne Stroustrup.
- Page 841. Alex Stepanov, 2003. Source: Bjarne Stroustrup.
- Page 927. Photo of diesel engine. Source: Mogens Hansen, MAN B&W, Copenhagen.
- Page 1023. AT&T Bell Labs' Murray Hill Research center, approx. 1990. Copyright: AT&T Bell Labs.





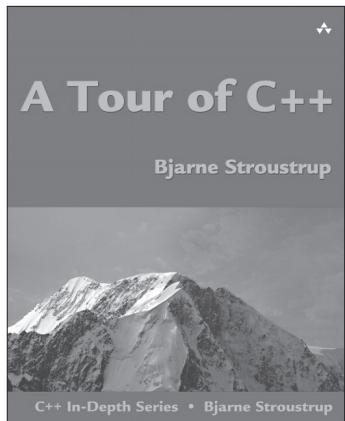
## More Guides from the Inventor of C++



ISBN-13: 978-0-321-56384-2

***The C++ Programming Language, Fourth Edition***, delivers meticulous, richly explained, and integrated coverage of the entire language—its facilities, abstraction mechanisms, standard libraries, and key design techniques. Throughout, Stroustrup presents concise, “pure C++11” examples, which have been carefully crafted to clarify both usage and program design.

Available in soft cover, hard cover, and eBook formats, and in Safari Books Online.



ISBN-13: 978-0-321-95831-0

In ***A Tour of C++***, Stroustrup excerpts the overview chapters from *The C++ Programming Language, Fourth Edition*, expanding and enhancing them to give an experienced programmer—in just a few hours—a clear idea of what constitutes modern C++. In this concise, self-contained guide, Stroustrup covers most major language features and the major standard-library components—not, of course, in great depth, but to a level that gives programmers a meaningful overview of the language, some key examples, and practical help in getting started.

Available in softcover and eBooks formats, and in Safari Books Online.



For more information and sample content visit  
[informit.com/stroustrup](http://informit.com/stroustrup)



# REGISTER



## THIS PRODUCT

[informit.com/register](http://informit.com/register)

Register the Addison-Wesley, Exam Cram, Prentice Hall, Que, and Sams products you own to unlock great benefits.

To begin the registration process, simply go to **informit.com/register** to sign in or create an account.

You will then be prompted to enter the 10- or 13-digit ISBN that appears on the back cover of your product.

Registering your products can unlock the following benefits:

- Access to supplemental content, including bonus chapters, source code, or project files.
- A coupon to be used on your next purchase.

Registration benefits vary by product. Benefits will be listed on your Account page under Registered Products.

### About InformIT — THE TRUSTED TECHNOLOGY LEARNING SOURCE

INFORMIT IS HOME TO THE LEADING TECHNOLOGY PUBLISHING IMPRINTS Addison-Wesley Professional, Cisco Press, Exam Cram, IBM Press, Prentice Hall Professional, Que, and Sams. Here you will gain access to quality and trusted content and resources from the authors, creators, innovators, and leaders of technology. Whether you're looking for a book on a new technology, a helpful article, timely newsletters, or access to the Safari Books Online digital library, InformIT has a solution for you.

# informIT.com

THE TRUSTED TECHNOLOGY LEARNING SOURCE

Addison-Wesley | Cisco Press | Exam Cram  
IBM Press | Que | Prentice Hall | Sams

SAFARI BOOKS ONLINE



**InformIT** is a brand of Pearson and the online presence for the world's leading technology publishers. It's your source for reliable and qualified content and knowledge, providing access to the top brands, authors, and contributors from the tech community.

▲Addison-Wesley

Cisco Press

EXAM/CRAM

IBM  
Press..

QUE

PRENTICE  
HALL

SAMS

Safari  
Books Online

## LearnIT at InformIT

Looking for a book, eBook, or training video on a new technology? Seeking timely and relevant information and tutorials? Looking for expert opinions, advice, and tips? **InformIT has the solution.**

- Learn about new releases and special promotions by subscribing to a wide variety of newsletters.  
Visit [informit.com/newsletters](#).
- Access FREE podcasts from experts at [informit.com/podcasts](#).
- Read the latest author articles and sample chapters at [informit.com/articles](#).
- Access thousands of books and videos in the Safari Books Online digital library at [safari.informit.com](#).
- Get tips from expert blogs at [informit.com/blogs](#).

Visit [informit.com/learn](#) to discover all the ways you can access the hottest technology content.

### Are You Part of the IT Crowd?

Connect with Pearson authors and editors via RSS feeds, Facebook, Twitter, YouTube, and more! Visit [informit.com/socialconnect](#).



# Try Safari Books Online FREE for 15 days

Get online access to Thousands of Books and Videos



**Safari**  
Books Online

FREE 15-DAY TRIAL + 15% OFF\*  
[informat.com/safaritrial](http://informat.com/safaritrial)

## ➤ Feed your brain

Gain unlimited access to thousands of books and videos about technology, digital media and professional development from O'Reilly Media, Addison-Wesley, Microsoft Press, Cisco Press, McGraw Hill, Wiley, WROX, Prentice Hall, Que, Sams, Apress, Adobe Press and other top publishers.

## ➤ See it, believe it

Watch hundreds of expert-led instructional videos on today's hottest topics.

## WAIT, THERE'S MORE!

## ➤ Gain a competitive edge

Be first to learn about the newest technologies and subjects with Rough Cuts pre-published manuscripts and new technology overviews in Short Cuts.

## ➤ Accelerate your project

Copy and paste code, create smart searches that let you know when new books about your favorite topics are available, and customize your library with favorites, highlights, tags, notes, mash-ups and more.

\* Available to new subscribers only. Discount applies to the Safari Library and is valid for first 12 consecutive monthly billing cycles. Safari Library is not available in all countries.



AdobePress

Cisco Press

FT  
FINANCIAL TIMES

IBM  
Press..

Microsoft  
Press

New  
Riders

O'REILLY



Peachpit  
Press

PEARSON  
IT Certification

PRENTICE  
HALL

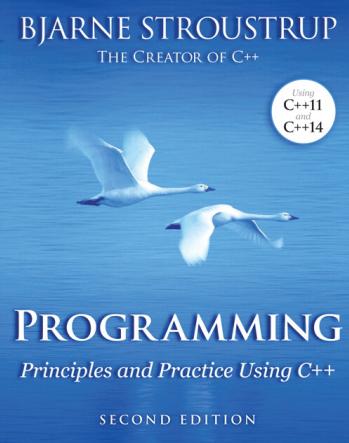
que

SAMS

VMWARE PRESS

WILEY

wro



**Safari**  
Books Online

# FREE Online Edition

Your purchase of **Programming, Second Edition**, includes access to a free online edition for 45 days through the **Safari Books Online** subscription service. Nearly every Addison-Wesley Professional book is available online through **Safari Books Online**, along with thousands of books and videos from publishers such as Cisco Press, Exam Cram, IBM Press, O'Reilly Media, Prentice Hall, Que, Sams, and VMware Press.

**Safari Books Online** is a digital library providing searchable, on-demand access to thousands of technology, digital media, and professional development books and videos from leading publishers. With one monthly or yearly subscription price, you get unlimited access to learning tools and information on topics including mobile app and software development, tips and tricks on using your favorite gadgets, networking, project management, graphic design, and much more.

Activate your FREE Online Edition at  
[informat.com/safarifree](http://informat.com/safarifree)

**STEP 1:** Enter the coupon code: QCXGNCB.

**STEP 2:** New Safari users, complete the brief registration form.  
Safari subscribers, just log in.

If you have difficulty registering on Safari or accessing the online edition,  
please e-mail [customer-service@safaribooksonline.com](mailto:customer-service@safaribooksonline.com)



Adobe Press



Cisco Press



O'REILLY



Peachpit  
Press



SAMS



VMWARE PRESS

