Appendix A

Programming Language Reference

This appendix is a compact description of the extended subset of C used for the examples in this book. It is *not* a complete description of C, much less of C++, which is an extended (some might say *over*extended) version of C for object-oriented programming. The C++ features (`class`, `public`, `private`) are used only in Part B. For complete descriptions and more explanation than we have room for here, see the Additional Reading.

A.1 Lexical Conventions

A.1.1 Comments

A comment in C is introduced by the characters `/*` and terminated by `*/`. In C++ (and in many implementations of C), a comment may also be introduced by the characters `//`, and this kind of comment is terminated by the next end-of-line. Comments do not occur within strings or character literals.

A.1.2 White Space

Blanks, tabs, new-lines, form-feeds, and comments are known as *white space*.

A.1.3 Preprocessing

Before a compiler translates source code to executable code, a preprocessor performs macro expansion and file inclusion, under the control of lines beginning with a `#` character (possibly after some white space), as follows.

A control line of the form

```
# define I X
```

where I is an identifier (Section A.1.5) and X is arbitrary text, causes the preprocessor to replace subsequent occurrences of the identifier with X. Leading and terminating white space are stripped from X.

A control line of the form

`# define` $I(I_0, I_1, \ldots, I_{n-1})\ X$

where I and the I_j are identifiers and there is no white space between I and the formal-parameter list, causes the processor to replace subsequent occurrences of $I(X_0, X_1, \ldots, X_{n-1})$ by X, but with occurrences of the macro parameters I_j in X replaced by the corresponding macro arguments X_j. Leading and terminating white space are stripped from X and from the arguments X_j.

The programs in this book assume the following macro definitions:

```
# define ASSERT(P)
# define FACT(P)
# define INVAR(P)
```

Because the macro bodies are null, calls of these macros are essentially comments.

A control line of the form

`# include` <*file name*>

causes the replacement of that line by the contents of the named file, which is searched for in a sequence of implementation-determined system directories. It is conventional to include the "headers" for standard libraries using this form.

A control line of the form

`# include` "*file name*"

causes the replacement of that line by the contents of the named file, which is searched for in a sequence of implementation-determined user directories, starting with the directory containing the source file. Included files are also preprocessed.

A control line of the form

`# ifndef` I

causes the preprocessor to exclude all program text until the following `# endif` line if the macro I is *not* defined. For example, the macro `__cplusplus` is defined if the processor is a C++ processor and is undefined if it is only a C processor, and so this facility may be used as in Program 3 in the Introduction to exclude a redundant definition of a `bool` type if C++ is in use.

If it is necessary to extend a control "line" over to an additional line, the last character in the line should be a backslash \ with the new-line character immediately following; the backslash and the new-line character are deleted.

A.1.4 Tokens

After preprocessing, a program consists of a sequence of tokens. There are six classes of tokens: identifiers, keywords, constants, string literals, operators, and other separators. White space is ignored, except that some white space is needed to separate otherwise adjacent identifiers, keywords, and constants.

A.1.5 Identifiers

An identifier is a sequence of letters, digits, or the underscore character (_). The first character must be a letter or underscore. Uppercase and lowercase letters are different. It is conventional to reserve identifiers whose first character is an underscore for use by standard libraries.

A.1.6 Keywords

The following identifiers are reserved for use as keywords in C:

`auto`	`do`	`goto`	`signed`	`unsigned`
`break`	`double`	`if`	`sizeof`	`void`
`case`	`else`	`int`	`static`	`volatile`
`char`	`enum`	`long`	`struct`	`while`
`const`	`extern`	`register`	`switch`	
`continue`	`float`	`return`	`typedef`	
`default`	`for`	`short`	`union`	

The following are additional keywords in C++:

`and`	`const_cast`	`namespace`	`public`	`typename`
`and_eq`	`delete`	`new`	`reinterpret_cast`	`using`
`asm`	`dynamic_cast`	`not`	`static_cast`	`virtual`
`bitand`	`explicit`	`not_eq`	`template`	`wchar_t`
`bitor`	`export`	`operator`	`this`	`xor`
`bool`	`false`	`or`	`throw`	`xor_eq`
`catch`	`friend`	`or_eq`	`true`	
`class`	`inline`	`private`	`try`	
`compl`	`mutable`	`protected`	`typeid`	

In particular, `class`, `private`, and `public` are *not* reserved in C and `bool`, `false`, and `true` *are* reserved in C++.

A.2 Basic Types and Constants

The basic data types used in this book and typical constants are given in the following table:

```
bool     false  true
char     'a'  'A'  ···  '\n'  '\\'  '\''  '\0'
int      ···  -2  -1  0  1  2  ···
float    0.  .1  3.14159  -1e2  6.625e-34  ···
```

The character designated by `\0` is termed the *null* character (the `NUL` in the ASCII character set). The character set actually used is implementation dependent. Type `bool` is predefined in C++, but it may be explicitly defined in C by the following declaration of an enumerated type (Section A.5):

```
typedef enum {false, true} bool;
```

Values of types `char` and `bool` are represented by small integer values, and the representations are not necessarily unique; for example, `false` is represented by 0, but *any* nonzero integer value will be taken as equivalent to `true` by `bool` operations. This means that the idiom *B* `== true` may be inappropriate for some `bool` expressions *B*. It is *not* illegal in C to apply `int` operations to `char` or `enum` variables, or even to `bool` operands, or vice versa; the programs in this book generally avoid such idioms.

Only finite ranges of values are representable by variables of type `int` or `char`. The standard `limits` library defines the following implementation-dependent constants:

`CHAR_MAX`	maximum value of type `char`
`CHAR_MIN`	minimum value of type `char`
`INT_MAX`	maximum value of type `int`
`INT_MIN`	minimum value of type `int`

There are also constants defining the properties of `float` values, but we will not describe these.

Values of type `int` are automatically converted to type `float` if necessary, but there may be round-off errors if an exact representation is not possible. If necessary, values of type `float` are converted to type `int` by discarding any fraction.

An attempt to compute an `int` value that is too large to be represented is illegal. This is called an *overflow*. The effect of an `int` overflow is not defined; many implementations simply ignore them, allowing garbage results to be produced. Type `int` may be replaced by `long int` (or just `long`), or even `long long int` in some implementations, to obtain a larger range of representable integers; similarly, type `float` may be replaced by `double`, or even `long double` in some implementations, to obtain greater precision or range.

A.3 Strings

A string constant consists of a sequence of characters (possibly none) enclosed in double-quote marks: `"..."`. Adjacent string constants are concatenated into a single string. After such concatenation, a null character is appended to the string to act as the string terminator. The escape sequences `\n` and `\"` for newline and double-quote characters, respectively, must be used if these characters are to be components of a string constant. A string is considered to be an array with `char` components. It is illegal to attempt to modify the representation of a string constant.

A.4 Variable Declarations

The basic form of a variable declaration is as follows:

- a type specifier: one of `char`, `int`, `float`, a defined type name (such as `bool`), an enumerated or structure type (Section A.5), or a class name (Section A.11), followed by
- one or more identifiers separated by commas and
- terminated with a semicolon (`;`) .

A declared variable may optionally be followed by `=` and an initializer expression that is assignment compatible with the variable; the variable is initialized to the value of the expression. If the type specifier is preceded by the qualifier `const`, the initial value may not subsequently be modified. If there is no explicit initializer, the initial value of a declared variable is unpredictable (i.e., garbage).

If a declared identifier is immediately followed by `[`K`]`, where K is a constant expression (Section A.6.4), the identifier is defined to be an *array* whose size is determined by the value of K; the size must be positive. Some implementations of C allow the size expression K to be an *arbitrary* (not necessarily constant) `int` expression. The array declarator may be optionally followed by `=` and a list of *constant* expressions enclosed in braces and separated by commas: `{`$K_0, K_1, \ldots, K_{n-1}$`}`. The list may also be terminated by a comma, if desired: `{`$K_0, K_1, \ldots, K_{n-1},$`}`. A `char` array may also be initialized by providing a string initializer of the form `"..."`. The number of initializers or the length of the string (plus 1 for the terminating null) must be $\leq K$ if a size expression K is specified, but the size expression may be omitted if an initializer list or string is supplied.

An n-dimensional array for $n > 1$ (i.e., an array of arrays) is declared by using n array specifiers `[`K_0`][`K_1`]` $\cdots$ `[`K_{n-1}`]`, optionally followed by `=` and a brace-enclosed and comma-separated (and possibly comma-terminated) list of constant initializer expressions. If an initializer list is provided, the *first* size specifier K_0 may be omitted; however, if K_0 is provided, the initializer list must have length ≤ the total number of components. Sublists may be enclosed by braces to indicate rows.

If a declared identifier is immediately *preceded* by `*`, the identifier is a *pointer* variable. In this book, this idiom is used only in Chapter 12 for file descriptors, as in the following:

```
FILE *f = tmpfile();
```

A.5 Enumerated and Structure Types, Defined Type Names

Enumerated types, structure types, and defined type names, as described next, may be used as type specifiers in declarations.

The construct `enum{`I_0`,`I_1`, ... ,`I_{n-1}`}` (where the I_j are identifiers) defines an *enumerated type*; each identifier I_j is thereby defined to be a constant with `int` value j. A variable declared to be of the enumerated type should only be assigned one of the enumerated values, but not every compiler enforces this.

The construct `struct{`$D_0\, D_1\, \cdots\, D_{n-1}$`}` (where each D_j is a variable declaration) defines a *structure type*. Each variable declared to be of the structure type will have n components (called *members* or *fields*), as specified in the declarations. The variable declarations D_j may *not* have initializers, not even constant expressions; however, a list of up to n initializing constant expressions may be provided in the declaration of a `struct` variable.

A declaration of the form `typedef` T I`;` (where T is a type and I is an identifier) defines I to be a new name for type T. A variable declared with a type-name specifier is considered to be of the type T given in the `typedef` declaration.

A.6 Expressions

The basic expression forms are constants, variables, function calls, operations, and parenthesized expressions.

A.6.1 Variable Expressions

An expression is considered to be a *variable* (known as an *l-value* expression in the C literature) if it is *either*

- an identifier declared to be a variable *or*
- of the form V`[`E`]` (where V is an array) *or*
- of the form V`.`I (where V is a structure or class object and I is one of its member variable names) *or*
- a parenthesized variable `(`V`)`.

It is illegal for the value of the subscript expression *E* to be outside the range determined by the size of the array when it was created; however, many implementations do not check this and the effects are unpredictable.

A.6.2 Function Calls

A function call normally consists of the identifier or class-object field selection (see Section A.11) designating the function, followed by a parenthesized (but possibly empty) list of argument expressions, separated by commas. The number of arguments must match the number expected by the function, and the corresponding types must be compatible. Function *definitions* are discussed in Section A.9.

A.6.3 Operations

The arithmetic operators are `+` and `-` and, with higher priority, `*`, `/`, and the remainder operator `%` (which should only be used on positive integers). Integer division discards any remainder. If all operands have type `int`, so does the result; otherwise, the operand values are converted to `float` and the result has type `float`.

The relational operators are `<`, `<=`, `>`, and `>=`, and, with lower priority, the equality operators `==` and `!=`. These operators all have lower priority than the arithmetic operators.

In C, assignment operations may be used as subexpressions; the programs in this book do not use this idiom. You will not get an error message if by accident you use `=` instead of `==` in an expression.

The `bool` operators are `!` for negation, `&&` for conjunction (i.e., and), and, with lower priority, `||` for disjunction (i.e., or). The latter two are evaluated "sequentially," so that the second operand is not evaluated if the value of the first operand determines the result. All these operations treat `0` as equivalent to `false` and any nonzero value as equivalent to `true`.

The `sizeof` operator yields the number of bytes required to store an object of the type of its operand, which may be either an expression or a parenthesized type. Note that, when `s` is an array parameter, `sizeof(s)` in the function is the size of an array reference (pointer), *not* the size of the actual array argument.

The address-of prefix operator `&` is needed to pass simple-variable arguments to functions by reference. In this book, this operator is used only for calls of the `scanf` function for formatted input (Section A.10). There is an inverse prefix operator for dereferencing (`*`), but it is not used in this book.

There are a number of operators that "shift" bit strings (`<<` and `>>`) or apply `bool` operations such as negation (`~`), conjunction (`&`), disjunction (`|`), and

Table A.1 Precedence and Associativity of Operators

`( ) [ ] .`	left
prefix operators: `! ~ + - sizeof & *`	right
`* / %`	left
`+ -`	left
`<< >>`	left
`< <= > >=`	left
`== !=`	left
`&`	left
`^`	left
`|`	left
`&&`	left
`||`	left
`? :`	right

exclusive-or (`^`) "bitwise" to bit strings. In this book, these are used only in the very stylized way described in Section 6.4.

Finally, there is the following ternary (three-operand) *conditional-expression* operator:

B `?` E_0 `:` E_1

where B normally has type `bool`. It is equivalent to

$$\begin{cases} E_0, & \text{if } B \texttt{ != false} \\ E_1, & \text{if } B \texttt{ == false} \end{cases}$$

Only one of E_0 or E_1 is evaluated. This operator associates to the *right* and has the lowest priority of all the operators we have discussed.

Table A.1 gives the operators (in order of precedence) and their associativity.

A.6.4 Constant Expressions

In certain contexts, expressions must be evaluated during compilation; these are termed *constant expressions*. They must not contain function calls, variables, array-subscripting operations, or structure-member selections, except in operands of the `sizeof` operator.

A.7 Some Library Functions

In this section, we briefly describe a selection of standard-library functions that return values and do not have side effects.

A.7.1 Mathematical Functions

Table A.2 summarizes the most useful "mathematical" functions available in the `math` library.

Table A.2 Mathematical Functions

`int abs(int i)`	absolute value of an `int`
`float fabs(float x)`	absolute value of a `float`
`int ceil(float x)`	smallest `int` not less than `x`
`int floor(float x)`	largest `int` not greater than `x`
`float sqrt(float x)`	$\sqrt{x}, x \geq 0$
`float log(float x)`	$\ln(x), x > 0$
`float log10(float x)`	$\log_{10}(x), x > 0$

A.7.2 Character Functions

Values `c` of type `char` may be classified by using the `bool` functions from the `ctype` library given in Table A.3. Also, the `char` functions `toupper(c)` and `tolower(c)` convert the case of `c` if it is a letter and return their argument unchanged if `c` is not a letter.

Table A.3 Character-Set Classification Functions

`bool isalpha(char c)`	letter
`bool isdigit(char c)`	decimal digit
`bool isupper(char c)`	uppercase letter
`bool islower(char c)`	lowercase letter
`bool iscntrl(char c)`	control character
`bool isprint(char c)`	printing character
`bool isgraph(char c)`	printing character except space
`bool ispunct(char c)`	printing character except space or letter or digit
`bool isspace(char c)`	space, form-feed, new-line, carriage return, tab

A.7.3 String Functions

The following is a selection of functions on strings defined in the `string` or `stdlib` libraries:

- `int strlen(const char[] s)`: length of string `s`, not including the terminating null (which must be present);

- `int strcmp(const char[] s, const char[] t)`: negative, `0`, or positive according to whether `s<t`, `s==t`, or `s>t`, respectively;
- `int strncmp(const char[] s, const char[] t, int n)`: same as `strcmp`, but compares at most `n` characters;
- `int atoi(const char[] s)`: converts string `s` to `int`;
- `float atof(const char[] s)`: converts string `s` to `float`;
- `int strspn(const char s[], const char t[])`: returns the length of the longest prefix of `s` consisting of characters in `t`;
- `int strcspn(const char s[], const char t[])`: returns the length of the longest prefix of `s` consisting of characters *not* in `t`.

A.8 Statements

Statements are executed for their effects and do not yield values.

A.8.1 Basic Statements

An *assignment statement* consists of an *assignment*, followed by a semicolon, where an assignment is one of the following forms:

- V `=` E where V is a variable expression and E is an expression whose value is convertible if necessary to the type of the variable;
- V *op*`=` E (where *op* is a suitable operator), which is equivalent to V `=` V *op* E except that the variable is only evaluated once;
- V`++`, which is equivalent to V `+= 1`;
- V`--`, which is equivalent to V `-= 1`.

Structures are assignable, but arrays (as a whole), including strings, are *not* assignable.

Assignments are actually *expressions* in C, but we do not use this idiom in this book. We have had to describe assignments (without the terminating semicolon) because they are used in the control section of the `for` loop (Section A.8.2).

A function call (Section A.6.2), followed by a semicolon is a statement. If the return type of the function is not `void`, the value returned is simply discarded.

The *null* statement consists of a semicolon by itself; it has no effect but may be used wherever a statement is needed syntactically.

A statement of the form `goto` I (where I is an identifier) transfers control to the statement labeled by identifier I (followed by a colon `:`) in the same function body. This feature is used in this book only in Exercise 8.4.

The `break;` and `return` E`;` (or, for a `void` function, just `return;`) statements may be used to exit loops and function bodies, respectively. In this book, we use these features only in very restricted ways.

A.8.2 Control Structures

A *compound* statement or *block* statement has the general form

`{` $D_0\ D_1 \cdots D_{n-1}\ C_0\ C_1 \cdots C_{m-1}$ `}`

where the D_i are declarations and the C_j are statements. The identifiers declared in a D_i have the rest of the block as their scope. The same identifier may not be declared more than once in the declarations of a block except that label identifiers and the member names for each `struct` type are considered to inhabit "name spaces" separate from the name space of identifiers for variables, functions, type names, and `enum` constants. Identifiers may be redeclared in nested blocks. Initializations in the declarations are performed each time the block is executed. The body of a function definition must be a block; however, in standard C a block may not itself contain a function definition. The trivial block `{}` has no effect and may be used wherever a statement is required.

The `if` statement forms are as follows:

- `if` (B) C
- `if (`B`)` C_0 `else` C_1

where B is a `bool` expression and C, C_0, and C_1 are statements. In combinations of the form `if (`B_0`) if (`B_1`)`C_0 `else` C_1, the `else` matches the immediately preceding unmatched `if` in the same block.

There is also a `switch` form, which is normally used as follows:

```
switch (N)
{ case K_0:
    C_0
    break;
  ⋮
  case K_i:
    C_i
    break;
  ⋮
  default:
    C_n
}
```

where N is an integer-valued expression, the K_i are constant integer-valued expressions with distinct values, and the C_i are (sequences of) statements. Expression N is evaluated, and control transfers to the C_i that is labeled by a constant expression with the same value (or to the `default`-labeled statement if none of the constants have that value). After execution of C_i, control is transferred by the terminating `break` to the end of the `switch` statement. If the `break` is omitted, however, control "falls through" to C_{i+1}; usually, this would not be what the programmer intends. Also, any of the C_i may have *several* `case` labels:

`case` K_{i0}`:`
`case` K_{i1}`:`
$\vdots$
`case` $K_{i(m-1)}$`:`
  C_i
  `break;`

The most basic form of *iteration* statement or *loop* has the following form:

`while (`B`)` C

where normally B is an expression of type `bool` and C is a statement (possibly, but not necessarily, a compound statement). The expression is evaluated before each execution of C; the loop terminates when the expression value has become equal to `false`, possibly before statement C is executed at all.

The `for` loop form

`for (`A_0`;` B`;` A_1`)` C

is equivalent to

A_0`; while (`B`){`C A_1`;}`

Here, A_0 and A_1 are assignments (i.e., assignment statements without terminating semicolons), B is normally an expression of type `bool`, and C is a statement. Any (or all) of A_0, B, or A_1 (but not the separating semicolons) may be omitted; if B is omitted, the condition is taken to be `true`.

The most common uses of the `for` form of loop are in the following "counting-up" and "counting-down" loops:

`for (`V`=`N_0`;`V`<`N_1`;`V`++)` C

`for (`V`=`N_0`;`V`>`N_1`;`V`--)` C

where V is a declared `int` variable and N_0 and N_1 are `int` expressions.

The following `do-while` form of loop is used when a loop condition is to be evaluated *after* each execution of the loop body (but not before the first execution):

`do` C `while (`B`);`

where C is a statement (usually a block) and B is normally a `bool` expression.

The `break;` statement may be used to exit from inside a loop. In this book, only the following special case is used:

```
for(;;)
{ D
  C0
  if (B) break;
  C1
}
```

where D is a declaration sequence (possibly empty), B is a `bool` expression, and C_0 and C_1 are statement sequences.

A.9 Function Definitions

Here is the basic form of a *function definition*:

$$T\ I(T_0\ I_0,\ T_1\ I_1,\ \ldots,T_{n-1}\ I_{n-1})\ C$$

where T (the return type) and the T_j (formal-parameter types) are types, I (the function name) and the I_j (the formal parameters) are identifiers, and C (the body) is a block. The formal parameters are considered to be defined in the block. If the function is intended to be used as a statement, the return type T should be `void`. The formal-parameter list may be either empty ($n = 0$) or `void` to indicate that the function does not require arguments; the enclosing parentheses are always required. A function may return a structure but may not return an array.

If the function is expected to return a value, the function body should be terminated by a statement of the form `return` E`;`, where E is an expression. In fact, `return` statements may be used anywhere in the function body, but this idiom is not used in this book, except in Exercise 8.4.

Parameter passing for simple variables is "by value"; that is, the value of an argument (actual parameter) is copied to a new local variable before the body is executed; assignments to the formal parameter do not affect the corresponding actual parameter.

If a formal-parameter identifier is immediately followed by an array specifier (`[`K`]` or `[]`), the *address* of the array is passed to the function; assignments to (components of) the formal parameter then affect the corresponding argument array. An array parameter may be preceded by the qualifier `const`; this qualification indicates that the array components will not be modified by the function. Multidimensional array parameters are treated in a similar manner.

If a formal-parameter identifier is immediately *preceded* by `*`, the corresponding argument must be a pointer, such as a file descriptor. The only other use of this idiom in this book is the standard `scanf` function (Section A.10); arguments of `scanf` must normally be prefixed by the `&` (address-of) operator.

Functions may be called recursively. If one function must call another function that has not yet been defined (perhaps because the two functions are *mutually* recursive), it is necessary to "declare" the called function, without *defining* it. A function *declaration* consists of a function *header* (type, name, formal-parameter list), followed by a semicolon (rather than a block). Examples of such declarations may be found in Programs 11.2 and 11.3.

A.10 More Library Functions

The library functions described in this section have side effects; some also return a value, though often the returned value is simply discarded.

A.10.1 Input and Output

The `stdio` library defines a type `FILE` of file descriptors. The identifiers `stdin`, `stdout`, and `stderr` are defined to be pointers to the file descriptors for the standard input, output, and error streams, respectively. Temporary files are created by calling the following function:

```
FILE *tmpfile(void)
```

which returns a pointer to the file descriptor, as in the following initialized variable declaration:

```
FILE *f = tmpfile();
```

Here are the basic input and output functions:

- `int getc(FILE *f)`: returns the `int` code for the next character in the file `f` or `EOF` (end-of-file) if there are no more characters to input;
- `int getchar(void)`: equivalent to `getc(stdin)`;
- `int putc(char c, FILE *f)`: appends `c` to file `f` and returns `EOF` if this fails;
- `int putchar(char c)`: equivalent to `putc(c, stdout)`;
- `int ungetc(char c, FILE *f)`: pushes `c` back into file `f` and returns `EOF` if this fails;
- `void rewind(FILE *f)`: resets the position of file `f` (not `stdin`, `stdout`, or `stderr`) to its first component;
- `void fgets(char s[], int n, FILE *f)`: reads at most the next `n-1` characters from `f` into `s`, up to and including a new-line; the string is then terminated by a null.

The constant `EOF` is defined by the `stdio` library; this is *not* a `char` value, which explains why the return types for these functions are `int` rather than `char`. Only one call of `ungetc` is allowed before the next `read` from that file. The character pushed back into the file need not be the same as the one previously read.

Formatted input and output are provided by the following functions:

- `int scanf(const char fmt[], ... )`
- `void printf(const char fmt[], ... )`
- `void fprintf(FILE *f, const char fmt[], ... )`

Here, `f` points to a file, and `fmt` is a format string that may contain *conversion specifications* to control conversions to or from the remaining arguments, as follows:

`%c`	`char`
`%i`	`int`
`%g`	`float`
`%s`	`char []`

For `scanf`, initial white space (including new-lines) is skipped for each conversion-specification item (except `c`). The input stream is then read up to the next white space and matched against the format item; however, if a number appears between the `%` and the control character in the conversion specification, it is used as the *maximum* width of the field read. It is a good idea to specify a maximum field width for string input to preclude buffer overflows. A string read using the `s` control character does not have to be quoted. Any remaining arguments of `scanf` should be prefixed by the "address of" operator `&` if they are simple (nonarray) variables. Other characters in the format string (i.e., those not escaped by `%`) must match the characters found in the input stream. Scanning of the input continues until the format string is exhausted or a match fails. The `int` returned by a call of `scanf` is the number of items successfully matched and assigned (or `EOF` if there isn't enough input); the value returned should always be checked to verify that the input was well formed, as in the following example:

```
if (scanf("%i,%i,%i", &a, &b, &c) != 3) error("input failure");
```

For `printf` and `fprintf`, a number used between the `%` and the control character is used as the *minimum* field width. A string argument is output up to the terminating null character. Other characters in the format string (including escape sequences such as `\n`) are output directly to the output stream without conversion. The function `fprintf` is similar but doesn't assume `stdout` (the standard output stream) as the default; we use this function to send error messages to the `stderr` stream, such as in the definition of function `error` on page 7.

Note that a call of the form `printf(str)` or `fprintf(f, str)` may have unexpected results if string `str` happens to contain the `%` character. The following are safe alternatives: `printf("%s", str)` and `fprintf(f, "%s", str)`.

A.10.2 More String Functions

The following function defined in the `string` library is normally executed for its effect.

> `void strncat(char s[], const char t[], int n)`: copies at most `n` characters from `t` to `s` starting at the terminating null of `s`; then `s` is padded if necessary with a single null.

The two string arguments should be distinct. There are standard functions in C to just copy (rather than concatenate) strings; unfortunately, they are unsafe or inefficient. The following function

```
void strlcpy(char s[], const char t[], int n)
/* copies at most n characters from t to s,
   up to and including the terminating null
*/
{ s[0] = '\0'; strncat(s, t, n); }
```

is efficient and safe, provided `n` is *smaller* than the size of array `s`.

A.10.3 Miscellaneous Functions

The function

```
int rand(void)
```

in the `stdlib` library returns a pseudo-random `int` in the range `0` to `RAND_MAX`.

The function

```
void exit(int status)
```

in `stdlib` aborts program execution, returning control to the system environment. The `status` argument is interpreted in a system-dependent way, but `EXIT_FAILURE` indicates unsuccessful termination and `EXIT_SUCCESS` or `0` indicate successful termination.

The `assert` library provides a function or macro

```
void assert(bool p)
```

that aborts execution with an error message if `p` evaluates to `false`.

A.11 Classes

Here is the typical form of a *class declaration* in C++:

```
class I
{ private: D_0 D_1 ··· D_{n-1}
  public: D'_0 D'_1 ··· D'_{m-1}
};
```

where I is an identifier (the class name), and the D_i and D'_j are declarations and function definitions. Note the terminating semicolon.

For any object V of type I and any I'_j declared in a *public* part of the `class` declaration, member I'_j is accessible using the notation $V\,.\,I'_j$; however, a *private* member I_i is accessible only *inside* the `class` declaration. According to the C++ standard, non-`static` variables declared in a `class` declaration may *not* have initializers, not even constant expressions. Class objects may be initialized by defining a (parameterless) *constructor* function with the same function name as the class; no return type should be specified for the constructor function. The constructor function for a class is automatically called for each class object created.

A.12 Program Structure

A complete program unit (after preprocessing) consists of a sequence of declarations and function definitions, including the definition of a function called `main`, which is the function that is, in effect, called by the system to start execution. The scope of identifiers declared at the program level is the rest of the program. Function `main` returns an `int` to the environment as a status indication; the interpretation is implementation dependent, but 0 conventionally indicates normal termination.

It is possible to separately compile program units; this capability is not really needed for the relatively small programs discussed here, but some of the associated features, such as use of the `static` qualifier and header files, are described in Section 5.4.

A.13 Grammar

This section presents a context-free grammar for the language described in this appendix. The notation is Backus-Naur formalism (BNF); it is explained in Section 10.1. Note that `true`, `false`, `bool`, `class`, `private`, and `public` are treated as *identifiers* in C and as *keywords* in C++. The following metavariables are not defined here: ⟨*identifier*⟩, ⟨*constant*⟩, and ⟨*string-literal*⟩. The grammar is actually ambiguous (Section 10.4); it is explained on page 237 how the nested-`if` ambiguity is resolved.

A.13.1 Expressions

```
⟨primary-expression⟩ ::= ⟨identifier⟩
                 |  ⟨constant⟩
                 |  ⟨string-literal⟩
                 |  ( ⟨expression⟩ )

⟨postfix-expression⟩ ::= ⟨primary-expression⟩
                 |  ⟨postfix-expression⟩ [ ⟨expression⟩ ]
                 |  ⟨postfix-expression⟩ ( )
                 |  ⟨postfix-expression⟩ ( ⟨expression-list⟩ )
                 |  ⟨postfix-expression⟩ . ⟨identifier⟩

⟨expression-list⟩ ::= ⟨expression⟩
                 |  ⟨expression-list⟩ , ⟨expression⟩

⟨unary-expression⟩ ::= ⟨postfix-expression⟩
                 |  ⟨unary-operator⟩ ⟨unary-expression⟩
                 |  sizeof ⟨unary-expression⟩
                 |  sizeof ( ⟨type-name⟩ )

⟨unary-operator⟩ ::= & | * | + | - | ~ | !

⟨multiplicative-expression⟩ ::= ⟨unary-expression⟩
                 |  ⟨multiplicative-expression⟩ * ⟨unary-expression⟩
                 |  ⟨multiplicative-expression⟩ / ⟨unary-expression⟩
                 |  ⟨multiplicative-expression⟩ % ⟨unary-expression⟩

⟨additive-expression⟩ ::= ⟨multiplicative-expression⟩
                 |  ⟨additive-expression⟩ + ⟨multiplicative-expression⟩
                 |  ⟨additive-expression⟩ - ⟨multiplicative-expression⟩

⟨shift-expression⟩ ::= ⟨additive-expression⟩
                 |  ⟨shift-expression⟩ << ⟨additive-expression⟩
                 |  ⟨shift-expression⟩ >> ⟨additive-expression⟩

⟨relational-expression⟩ ::= ⟨shift-expression⟩
                 |  ⟨relational-expression⟩ < ⟨shift-expression⟩
                 |  ⟨relational-expression⟩ > ⟨shift-expression⟩
                 |  ⟨relational-expression⟩ <= ⟨shift-expression⟩
                 |  ⟨relational-expression⟩ >= ⟨shift-expression⟩

⟨equality-expression⟩ ::= ⟨relational-expression⟩
                 |  ⟨equality-expression⟩ == ⟨relational-expression⟩
                 |  ⟨equality-expression⟩ != ⟨relational-expression⟩

⟨and-expression⟩ ::= ⟨equality-expression⟩
                 |  ⟨and-expression⟩ & ⟨equality-expression⟩
```

```
⟨exclusive-or-expression⟩ ::= ⟨and-expression⟩
                 |  ⟨exclusive-or-expression⟩ ^ ⟨and-expression⟩

⟨inclusive-or-expression⟩ ::= ⟨exclusive-or-expression⟩
                 |  ⟨inclusive-or-expression⟩ | ⟨exclusive-or-expression⟩

⟨logical-and-expression⟩ ::= ⟨inclusive-or-expression⟩
                 |  ⟨logical-and-expression⟩ && ⟨inclusive-or-expression⟩

⟨logical-or-expression⟩ ::= ⟨logical-and-expression⟩
                 |  ⟨logical-or-expression⟩ || ⟨logical-and-expression⟩

⟨expression⟩ ::= ⟨logical-or-expression⟩
                 |  ⟨logical-or-expression⟩ ? ⟨expression⟩ : ⟨expression⟩

⟨constant-expression⟩ ::= ⟨expression⟩
```

A.13.2 Statements

```
⟨statement⟩ ::= ⟨labeled-statement⟩
                 |  ⟨compound-statement⟩
                 |  ⟨assign-statement⟩
                 |  ⟨selection-statement⟩
                 |  ⟨iteration-statement⟩
                 |  ⟨jump-statement⟩

⟨labeled-statement⟩ ::= ⟨identifier⟩ : ⟨statement⟩
                 |  case ⟨constant-expression⟩ : ⟨statement⟩
                 |  default : ⟨statement⟩

⟨compound-statement⟩ ::= { }
                 |  { ⟨statement-list⟩ }
                 |  { ⟨declaration-list⟩ }
                 |  { ⟨declaration-list⟩ ⟨statement-list⟩ }

⟨declaration-list⟩ ::= ⟨block-declaration⟩
                 |  ⟨declaration-list⟩ ⟨block-declaration⟩

⟨statement-list⟩ ::= ⟨statement⟩
                 |  ⟨statement-list⟩ ⟨statement⟩

⟨assign-statement⟩ ::= ;
                 |  ⟨assignment⟩ ;

⟨assignment⟩ ::= ⟨postfix-expression⟩ ⟨assignment-operator⟩ ⟨expression⟩
                 |  ⟨postfix-expression⟩ ++
                 |  ⟨postfix-expression⟩ --

⟨assignment-operator⟩ ::= = | *= | /= | %= | += | -= | <<= | >>= | &= | |= | ^=
```

⟨*selection-statement*⟩ ::= `if` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩
  | `if` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩ `else` ⟨*statement*⟩
  | `switch` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩

⟨*iteration-statement*⟩ ::= `while` `(` ⟨*expression*⟩ `)` ⟨*statement*⟩
  | `do` ⟨*statement*⟩ `while` `(` ⟨*expression*⟩ `)` `;`
  | `for` `(` ⟨*assign-statement*⟩ `;` `)` ⟨*statement*⟩
  | `for` `(` ⟨*assign-statement*⟩ ⟨*expression*⟩ `;` `)` ⟨*statement*⟩
  | `for` `(` ⟨*assign-statement*⟩ ⟨*expression*⟩ `;` ⟨*assignment*⟩ `)` ⟨*statement*⟩

⟨*jump-statement*⟩ ::= `goto` ⟨*identifier*⟩ `;`
  | `break` `;`
  | `return` `;`
  | `return` ⟨*expression*⟩ `;`

A.13.3 Declarations

⟨*declaration*⟩ ::= ⟨*function-definition*⟩
  | ⟨*block-declaration*⟩

⟨*function-definition*⟩ ::= ⟨*declaration-specifiers*⟩ ⟨*declarator*⟩ ⟨*compound-statement*⟩
  | ⟨*declarator*⟩ ⟨*compound-statement*⟩

⟨*block-declaration*⟩ ::= ⟨*declaration-specifiers*⟩ `;`
  | ⟨*declaration-specifiers*⟩ ⟨*init-declarator-list*⟩ `;`

⟨*init-declarator-list*⟩ ::= ⟨*init-declarator*⟩
  | ⟨*init-declarator-list*⟩ `,` ⟨*init-declarator*⟩

⟨*init-declarator*⟩ ::= ⟨*declarator*⟩
  | ⟨*declarator*⟩ `=` ⟨*initializer*⟩

⟨*initializer*⟩ ::= ⟨*expression*⟩
  | `{` ⟨*initializer-list*⟩ `}`
  | `{` ⟨*initializer-list*⟩ `,` `}`

⟨*initializer-list*⟩ ::= ⟨*initializer*⟩
  | ⟨*initializer-list*⟩ `,` ⟨*initializer*⟩

⟨*declaration-specifiers*⟩ ::= ⟨*declaration-specifier*⟩
  | ⟨*declaration-specifier*⟩ ⟨*declaration-specifiers*⟩

⟨*declaration-specifier*⟩ ::= `const`
  | `static`
  | `typedef`
  | ⟨*type-specifier*⟩

```
⟨type-specifier⟩ ::= void
               |  char
               |  short
               |  int
               |  long
               |  float
               |  double
               |  unsigned
               |  enum { ⟨identifier-list⟩ }
               |  struct { ⟨member-declaration-list⟩ }
               |  class ⟨identifier⟩ { ⟨member-declaration-list⟩ }
               |  ⟨type-name⟩

⟨identifier-list⟩ ::= ⟨identifier⟩
               |  ⟨identifier-list⟩ , ⟨identifier⟩

⟨member-declaration-list⟩ ::= ⟨member-declaration⟩
               |  ⟨member-declaration-list⟩ ⟨member-declaration⟩
               |  private : ⟨member-declaration-list⟩
               |  public : ⟨member-declaration-list⟩

⟨member-declaration⟩ ::= ⟨declaration-specifiers⟩ ⟨member-declarator-list⟩ ;
               |  ⟨function-definition⟩
               |  ⟨function-definition⟩ ;

⟨member-declarator-list⟩ ::= ⟨declarator⟩
               |  ⟨member-declarator-list⟩ , ⟨declarator⟩

⟨declarator⟩ ::= * ⟨direct-declarator⟩
               |  ⟨direct-declarator⟩

⟨direct-declarator⟩ ::= ⟨identifier⟩
               |  ( ⟨declarator⟩ )
               |  ⟨direct-declarator⟩ [ ⟨constant-expression⟩ ]
               |  ⟨direct-declarator⟩ [ ]
               |  ⟨direct-declarator⟩ ( ⟨parameter-list⟩ )
               |  ⟨direct-declarator⟩ ( )

⟨parameter-list⟩ ::= ⟨parameter-declaration⟩
               |  ⟨parameter-list⟩ , ⟨parameter-declaration⟩

⟨parameter-declaration⟩ ::= ⟨declaration-specifiers⟩ ⟨declarator⟩
               |  ⟨declaration-specifiers⟩ ⟨abstract-declarator⟩
               |  ⟨declaration-specifiers⟩

⟨type-name⟩ ::= ⟨declaration-specifiers⟩
               |  ⟨declaration-specifiers⟩ ⟨abstract-declarator⟩
```

$\langle$*abstract-declarator*$\rangle$ ::= `*`
  | $\langle$*direct-abstract-declarator*$\rangle$
  | `*` $\langle$*direct-abstract-declarator*$\rangle$

$\langle$*direct-abstract-declarator*$\rangle$ ::= `(` $\langle$*abstract-declarator*$\rangle$ `)`
  | `[ ]`
  | `[` $\langle$*constant-expression*$\rangle$ `]`
  | $\langle$*direct-abstract-declarator*$\rangle$ `[ ]`
  | $\langle$*direct-abstract-declarator*$\rangle$ `[` $\langle$*constant-expression*$\rangle$ `]`
  | `( )`
  | `(` $\langle$*parameter-list*$\rangle$ `)`
  | $\langle$*direct-abstract-declarator*$\rangle$ `( )`
  | $\langle$*direct-abstract-declarator*$\rangle$ `(` $\langle$*parameter-list*$\rangle$ `)`

A.13.4 Programs

$\langle$*program*$\rangle$ ::= $\langle$*declaration*$\rangle$
  | $\langle$*program*$\rangle$ $\langle$*declaration*$\rangle$

A.14 Additional Reading

The C programming language as of 1988 is described in detail in [KR88]. Online textbooks are available here:

`http://www.eskimo.com/~scs/cclass/index.html`

`http://www.strath.ac.uk/CC/Courses/NewCcourse/ccourse.html`

The standard C libraries are described in detail here:

`http://secure.dinkumware.com/htm_cl/index.html`

A "rationale" for the 1989 ANSI C standard may be found here:

`http://www.lysator.liu.se/c/rat/title.html`

A context-free grammar for all of C may be found here:

`http://www.lysator.liu.se/c/ANSI-C-grammar-y.html`

The lexical aspects are specified by (extended) regular expressions here:

`http://www.lysator.liu.se/c/ANSI-C-grammar-l.html`

A detailed description of C++ as of 1997 and discussions of programming style in C++ may be found in [Str97].

References

[KR88] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*, 2nd edition. Prentice Hall, 1988.

[Str97] B. Stroustrup. *The C++ Programming Language*, 3rd edition. Addison-Wesley, 1997.