6

Expressions and Statements

Premature optimization
is the root of all evil.
– D. Knuth

On the other hand,
we cannot ignore efficiency.
– Jon Bentley

Desk calculator example — input — command line arguments — expression summary — logical and relational operators — increment and decrement — free store — explicit type conversion — statement summary — declarations — selection statements — declarations in conditions — iteration statements — the infamous ***goto*** — comments and indentation — advice — exercises.

6.1 A Desk Calculator [expr.calculator]

Statements and expressions are introduced by presenting a desk calculator program that provides the four standard arithmetic operations as infix operators on floating-point numbers. The user can also define variables. For example, given the input

```
r = 2.5
area = pi * r * r
```

(pi is predefined) the calculator program will write

```
2.5
19.635
```

where `2.5` is the result of the first line of input and `19.635` is the result of the second.

The calculator consists of four main parts: a parser, an input function, a symbol table, and a driver. Actually, it is a miniature compiler in which the parser does the syntactic analysis, the input function handles input and lexical analysis, the symbol table holds permanent information, and the driver handles initialization, output, and errors. We could add many features to this calculator to make it more useful (§6.6[20]), but the code is long enough as it is, and most features would just add code without providing additional insight into the use of C++.

6.1.1 The Parser [expr.parser]

Here is a grammar for the language accepted by the calculator:

```
program:
        END                             // END is end-of-input
        expr_list END

expr_list:
        expression PRINT                // PRINT is semicolon
        expression PRINT expr_list

expression:
        expression + term
        expression - term
        term

term:
        term / primary
        term * primary
        primary

primary:
        NUMBER
        NAME
        NAME = expression
        - primary
        ( expression )
```

In other words, a program is a sequence of expressions separated by semicolons. The basic units of an expression are numbers, names, and the operators `*`, `/`, `+`, `-` (both unary and binary), and `=`. Names need not be declared before use.

The style of syntax analysis used is usually called *recursive descent*; it is a popular and straightforward top-down technique. In a language such as C++, in which function calls are relatively cheap, it is also efficient. For each production in the grammar, there is a function that calls other functions. Terminal symbols (for example, ***END***, ***NUMBER***, `+`, and `-`) are recognized by the lexical analyzer, ***get_token***`()`; and nonterminal symbols are recognized by the syntax analyzer functions, ***expr***`()`, ***term***`()`, and ***prim***`()`. As soon as both operands of a (sub)expression are known, the expression is evaluated; in a real compiler, code could be generated at this point.

The parser uses a function ***get_token***`()` to get input. The value of the most recent call of ***get_token***`()` can be found in the global variable ***curr_tok***. The type of ***curr_tok*** is the enumeration ***Token_value***:

```
enum Token_value {
    NAME,       NUMBER,     END,
    PLUS='+',   MINUS='-',  MUL='*',    DIV='/',
    PRINT=';',  ASSIGN='=', LP='(',     RP=')'
};

Token_value curr_tok = PRINT;
```

Representing each token by the integer value of its character is convenient and efficient and can be a help to people using debuggers. This works as long as no character used as input has a value used as an enumerator – and no character set I know of has a printing character with a single-digit integer value. I chose ***PRINT*** as the initial value for ***curr_tok*** because that is the value it will have after the calculator has evaluated an expression and displayed its value. Thus, I ''start the system'' in a normal state to minimize the chance of errors and the need for special startup code.

Each parser function takes a ***bool*** (§4.2) argument indicating whether the function needs to call ***get_token()*** to get the next token. Each parser function evaluates ''its'' expression and returns the value. The function ***expr()*** handles addition and subtraction. It consists of a single loop that looks for terms to add or subtract:

```
double expr(bool get)           // add and subtract
{
    double left = term(get);

    for (;;)                    // ''forever''
        switch (curr_tok) {
        case PLUS:
            left += term(true);
            break;
        case MINUS:
            left -= term(true);
            break;
        default:
            return left;
        }
}
```

This function really does not do much itself. In a manner typical of higher-level functions in a large program, it calls other functions to do the work.

The *switch-statement* tests the value of its condition, which is supplied in parentheses after the ***switch*** keyword, against a set of constants. The *break-statement*s are used to exit the *switch-statement*. The constants following the ***case*** labels must be distinct. If the value tested does not match any ***case*** label, the ***default*** is chosen. The programmer need not provide a ***default***.

Note that an expression such as `2-3+4` is evaluated as `(2-3)+4`, as specified in the grammar.

The curious notation ***for(;;)*** is the standard way to specify an infinite loop; you could pronounce it ''forever.'' It is a degenerate form of a *for-statement* (§6.3.3); ***while(true)*** is an alternative. The *switch-statement* is executed repeatedly until something different from `+` and `-` is found, and then the *return-statement* in the default case is executed.

The operators `+=` and `-=` are used to handle the addition and subtraction; ***left=left+term()*** and

left=left-term( ) could have been used without changing the meaning of the program. However, *left+=term*( ) and *left-=term*( ) not only are shorter but also express the intended operation directly. Each assignment operator is a separate lexical token, so *a + = 1;* is a syntax error because of the space between the + and the =.

Assignment operators are provided for the binary operators

```
+   -   *   /   %   &   |   ^   <<  >>
```

so that the following assignment operators are possible

```
=   +=  -=  *=  /=  %=  &=  |=  ^=  <<= >>=
```

The % is the modulo, or remainder, operator; &, |, and ^ are the bitwise logical operators AND, OR, and exclusive OR; << and >> are the left shift and right shift operators; §6.2 summarizes the operators and their meanings. For a binary operator @ applied to operands of built-in types, an expression *x@=y* means *x=x@y*, except that *x* is evaluated once only.

Chapter 8 and Chapter 9 discuss how to organize a program as a set of modules. With one exception, the declarations for this calculator example can be ordered so that everything is declared exactly once and before it is used. The exception is *expr*( ), which calls *term*( ), which calls *prim*( ), which in turn calls *expr*( ). This loop must be broken somehow. A declaration

```
double expr(bool);
```

before the definition of *prim*( ) will do nicely.

Function *term*( ) handles multiplication and division in the same way *expr*( ) handles addition and subtraction:

```
double term(bool get)           // multiply and divide
{
    double left = prim(get);

    for (;;)
        switch (curr_tok) {
        case MUL:
            left *= prim(true);
            break;
        case DIV:
            if (double d = prim(true)) {
                left /= d;
                break;
            }
            return error("divide by 0");
        default:
            return left;
        }
}
```

The result of dividing by zero is undefined and usually disastrous. We therefore test for *0* before dividing and call *error*( ) if we detect a zero divisor. The function *error*( ) is described in §6.1.4.

The variable *d* is introduced into the program exactly where it is needed and initialized immediately. The scope of a name introduced in a condition is the statement controlled by that condition,

and the resulting value is the value of the condition (§6.3.2.1). Consequently, the division and assignment ***left/=d*** is done if and only if ***d*** is nonzero.

The function ***prim()*** handling a *primary* is much like ***expr()*** and ***term()***, except that because we are getting lower in the call hierarchy a bit of real work is being done and no loop is necessary:

```
double number_value;
string string_value;

double prim(bool get)          // handle primaries
{
    if (get) get_token();

    switch (curr_tok) {
    case NUMBER:               // floating-point constant
    {   double v = number_value;
        get_token();
        return v;
    }
    case NAME:
    {   double& v = table[string_value];
        if (get_token() == ASSIGN) v = expr(true);
        return v;
    }
    case MINUS:            // unary minus
        return -prim(true);
    case LP:
    {   double e = expr(true);
        if (curr_tok != RP) return error(") expected");
        get_token();           // eat ')'
        return e;
    }
    default:
        return error("primary expected");
    }
}
```

When a ***NUMBER*** (that is, an integer or floating-point literal) is seen, its value is returned. The input routine ***get_token()*** places the value in the global variable ***number_value***. Use of a global variable in a program often indicates that the structure is not quite clean – that some sort of optimization has been applied. So it is here. Ideally, a lexical token consists of two parts: a value specifying the kind of token (a ***Token_value*** in this program) and (when needed) the value of the token. Here, there is only a single, simple variable, ***curr_tok***, so the global variable ***number_value*** is needed to hold the value of the last ***NUMBER*** read. Eliminating this spurious global variable is left as an exercise (§6.6[21]). Saving the value of ***number_value*** in the local variable ***v*** before calling ***get_token()*** is not really necessary. For every legal input, the calculator always uses one number in the computation before reading another from input. However, saving the value and displaying it correctly after an error helps the user.

In the same way that the value of the last ***NUMBER*** is kept in ***number_value***, the character string representation of the last ***NAME*** seen is kept in ***string_value***. Before doing anything to a

name, the calculator must first look ahead to see if it is being assigned to or simply read. In both cases, the symbol table is consulted. The symbol table is a ***map*** (§3.7.4, §17.4.1):

```
map<string,double> table;
```

That is, when ***table*** is indexed by a ***string***, the resulting value is the ***double*** corresponding to the ***string***. For example, if the user enters

```
radius = 6378.388;
```

the calculator will execute

```
double& v = table["radius"];
// ... expr() calculates the value to be assigned ...
v = 6378.388;
```

The reference ***v*** is used to hold on to the ***double*** associated with ***radius*** while ***expr()*** calculates the value ***6378.388*** from the input characters.

6.1.2 The Input Function [expr.input]

Reading input is often the messiest part of a program. This is because a program must communicate with a person, it must cope with that person's whims, conventions, and seemingly random errors. Trying to force the person to behave in a manner more suitable for the machine is often (rightly) considered offensive. The task of a low-level input routine is to read characters and compose higher-level tokens from them. These tokens are then the units of input for higher-level routines. Here, low-level input is done by ***get_token()***. Writing a low-level input routine need not be an everyday task. Many systems provide standard functions for this.

I build ***get_token()*** in two stages. First, I provide a deceptively simple version that imposes a burden on the user. Next, I modify it into a slightly less elegant, but much easier to use, version.

The idea is to read a character, use that character to decide what kind of token needs to be composed, and then return the ***Token_value*** representing the token read.

The initial statements read the first non-whitespace character into ***ch*** and check that the read operation succeeded:

```
Token_value get_token()
{
    char ch = 0;
    cin>>ch;

    switch (ch) {
    case 0:
        return curr_tok=END;      // assign and return
```

By default, operator >> skips whitespace (that is, spaces, tabs, newlines, etc.) and leaves the value of ***ch*** unchanged if the input operation failed. Consequently, ***ch==0*** indicates end of input.

Assignment is an operator, and the result of the assignment is the value of the variable assigned to. This allows me to assign the value ***END*** to ***curr_tok*** and return it in the same statement. Having a single statement rather than two is useful in maintenance. If the assignment and the return became separated in the code, a programmer might update the one and forget to update to the other.

Let us look at some of the cases separately before considering the complete function. The expression terminator ´;´, the parentheses, and the operators are handled simply by returning their values:

```
case ´;´:
case ´*´:
case ´/´:
case ´+´:
case ´-´:
case ´(´:
case ´)´:
case ´=´:
    return curr_tok=Token_value(ch);
```

Numbers are handled like this:

```
case ´0´: case ´1´: case ´2´: case ´3´: case ´4´:
case ´5´: case ´6´: case ´7´: case ´8´: case ´9´:
case ´.´:
    cin.putback(ch);
    cin >> number_value;
    return curr_tok=NUMBER;
```

Stacking ***case*** labels horizontally rather than vertically is generally not a good idea because this arrangement is harder to read. However, having one line for each digit is tedious. Because operator >> is already defined for reading floating-point constants into a ***double***, the code is trivial. First the initial character (a digit or a dot) is put back into ***cin***. Then the constant can be read into ***number_value***.

A name is handled similarly:

```
default:                // NAME, NAME =, or error
    if (isalpha(ch)) {
        cin.putback(ch);
        cin>>string_value;
        return curr_tok=NAME;
    }
    error("bad token");
    return curr_tok=PRINT;
```

The standard library function ***isalpha()*** (§20.4.2) is used to avoid listing every character as a separate ***case*** label. Operator >> applied to a string (in this case, ***string_value***) reads until it hits white-space. Consequently, a user must terminate a name by a space before an operator using the name as an operand. This is less than ideal, so we will return to this problem in §6.1.3.

Here, finally, is the complete input function:

```
Token_value get_token()
{
    char ch = 0;
    cin>>ch;
```

```
        switch (ch) {
        case 0:
                return curr_tok=END;
        case ´;´:
        case ´*´:
        case ´/´:
        case ´+´:
        case ´-´:
        case ´(´:
        case ´)´:
        case ´=´:
                return curr_tok=Token_value(ch);
        case ´0´: case ´1´: case ´2´: case ´3´: case ´4´:
        case ´5´: case ´6´: case ´7´: case ´8´: case ´9´:
        case ´.´:
                cin.putback(ch);
                cin >> number_value;
                return curr_tok=NUMBER;
        default:                        // NAME, NAME =, or error
                if (isalpha(ch)) {
                        cin.putback(ch);
                        cin>>string_value;
                        return curr_tok=NAME;
                }
                error("bad token");
                return curr_tok=PRINT;
        }
}
```

The conversion of an operator to its token value is trivial because the ***Token_value*** of an operator was defined as the integer value of the operator (§4.8).

6.1.3 Low-level Input [expr.low]

Using the calculator as defined so far reveals a few inconveniences. It is tedious to remember to add a semicolon after an expression in order to get its value printed, and having a name terminated by whitespace only is a real nuisance. For example, *x=7* is an identifier – rather than the identifier *x* followed by the operator = and the number *7*. Both problems are solved by replacing the type-oriented default input operations in ***get_token()*** with code that reads individual characters.

First, we'll make a newline equivalent to the semicolon used to mark the end of expression:

```
Token_value get_token()
{
        char ch;

        do { // skip whitespace except '\n'
                if(!cin.get(ch)) return curr_tok = END;
        } while (ch!=´\n´ && isspace(ch));
```

```
switch (ch) {
case ';':
case '\n':
        return curr_tok=PRINT;
```

A *do-statement* is used; it is equivalent to a *while-statement* except that the controlled statement is always executed at least once. The call ***cin.get(ch)*** reads a single character from the standard input stream into ***ch***. By default, ***get()*** does not skip whitespace the way ***operator >>*** does. The test ***if (!cin.get(ch))*** fails if no character can be read from ***cin***; in this case, ***END*** is returned to terminate the calculator session. The operator ! (NOT) is used because ***get()*** returns ***true*** in case of success.

The standard library function ***isspace()*** provides the standard test for whitespace (§20.4.2); ***isspace(c)*** returns a nonzero value if ***c*** is a whitespace character and zero otherwise. The test is implemented as a table lookup, so using ***isspace()*** is much faster than testing for the individual whitespace characters. Similar functions test if a character is a digit – ***isdigit()*** – a letter – ***isalpha()*** – or a digit or letter – ***isalnum()***.

After whitespace has been skipped, the next character is used to determine what kind of lexical token is coming.

The problem caused by >> reading into a string until whitespace is encountered is solved by reading one character at a time until a character that is not a letter or a digit is found:

```
default:                    // NAME, NAME=, or error
        if (isalpha(ch)) {
                string_value = ch;
                while (cin.get(ch) && isalnum(ch)) string_value.push_back(ch);
                cin.putback(ch);
                return curr_tok=NAME;
        }
        error("bad token");
        return curr_tok=PRINT;
```

Fortunately, these two improvements could both be implemented by modifying a single local section of code. Constructing programs so that improvements can be implemented through local modifications only is an important design aim.

6.1.4 Error Handling [expr.error]

Because the program is so simple, error handling is not a major concern. The error function simply counts the errors, writes out an error message, and returns:

```
int no_of_errors;

double error(const string& s)
{
        no_of_errors++;
        cerr << "error: " << s << '\n';
        return 1;
}
```

The stream ***cerr*** is an unbuffered output stream usually used to report errors (§21.2.1).

The reason for returning a value is that errors typically occur in the middle of the evaluation of an expression, so we should either abort that evaluation entirely or return a value that is unlikely to cause subsequent errors. The latter is adequate for this simple calculator. Had *get_token*() kept track of the line numbers, *error*() could have informed the user approximately where the error occurred. This would be useful when the calculator is used noninteractively (§6.6[19]).

Often, a program must be terminated after an error has occurred because no sensible way of continuing has been devised. This can be done by calling *exit*(), which first cleans up things like output streams and then terminates the program with its argument as the return value (§9.4.1.1).

More stylized error-handling mechanisms can be implemented using exceptions (see §8.3, Chapter 14), but what we have here is quite suitable for a 150-line calculator.

6.1.5 The Driver [expr.driver]

With all the pieces of the program in place, we need only a driver to start things. In this simple example, *main*() can do that:

```
int main()
{
    table["pi"] = 3.1415926535897932385;    // insert predefined names
    table["e"] = 2.7182818284590452354;

    while (cin) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
    }

    return no_of_errors;
}
```

Conventionally, *main*() should return zero if the program terminates normally and nonzero otherwise (§3.2). Returning the number of errors accomplishes this nicely. As it happens, the only initialization needed is to insert the predefined names into the symbol table.

The primary task of the main loop is to read expressions and write out the answer. This is achieved by the line:

```
cout << expr(false) << '\n';
```

The argument *false* tells *expr*() that it does not need to call *get_token*() to get a current token on which to work.

Testing *cin* each time around the loop ensures that the program terminates if something goes wrong with the input stream, and testing for *END* ensures that the loop is correctly exited when *get_token*() encounters end-of-file. A *break-statement* exits its nearest enclosing *switch-statement* or loop (that is, a *for-statement*, *while-statement*, or *do-statement*). Testing for *PRINT* (that is, for '\n' and ';') relieves *expr*() of the responsibility for handling empty expressions. A *continue-statement* is equivalent to going to the very end of a loop, so in this case

```
while (cin) {
        // ...
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
}
```

is equivalent to

```
while (cin) {
        // ...
        if (curr_tok != PRINT)
                cout << expr(false) << '\n';
}
```

6.1.6 Headers [expr.headers]

The calculator uses standard library facilities. Therefore, appropriate headers must be #**include**d to complete the program:

```
#include<iostream>    // I/O
#include<string>      // strings
#include<map>         // map
#include<cctype>      // isalpha(), etc.
```

All of these headers provide facilities in the **std** namespace, so to use the names they provide we must either use explicit qualification with **std**:: or bring the names into the global namespace by

```
using namespace std;
```

To avoid confusing the discussion of expressions with modularity issues, I did the latter. Chapter 8 and Chapter 9 discuss ways of organizing this calculator into modules using namespaces and how to organize it into source files. On many systems, standard headers have equivalents with a **.h** suffix that declare the classes, functions, etc., and place them in the global namespace (§9.2.1, §9.2.4, §B.3.1).

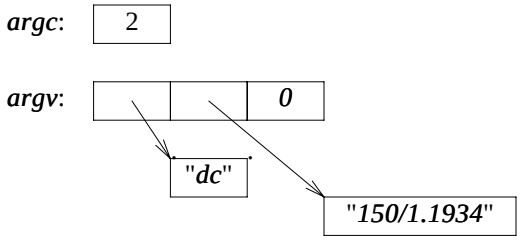
6.1.7 Command-Line Arguments [expr.command]

After the program was written and tested, I found it a bother to first start the program, then type the expressions, and finally quit. My most common use was to evaluate a single expression. If that expression could be presented as a command-line argument, a few keystrokes could be avoided.

A program starts by calling **main**() (§3.2, §9.4). When this is done, **main**() is given two arguments specifying the number of arguments, usually called **argc**, and an array of arguments, usually called **argv**. The arguments are character strings, so the type of **argv** is **char***[**argc+1**]. The name of the program (as it occurs on the command line) is passed as **argv**[**0**], so **argc** is always at least **1**. The list of arguments is zero-terminated; that is, **argv**[**argc**]==**0**. For example, for the command

```
dc 150/1.1934
```

the arguments have these values:



Because the conventions for calling ***main()*** are shared with C, C-style arrays and strings are used.

It is not difficult to get hold of a command-line argument. The problem is how to use it with minimal reprogramming. The idea is to read from the command string in the same way that we read from the input stream. A stream that reads from a string is unsurprisingly called an ***istringstream***. Unfortunately, there is no elegant way of making ***cin*** refer to an ***istringstream***. Therefore, we must find a way of getting the calculator input functions to refer to an ***istringstream***. Furthermore, we must find a way of getting the calculator input functions to refer to an ***istringstream*** or to ***cin*** depending on what kind of command-line argument we supply.

A simple solution is to introduce a global pointer ***input*** that points to the input stream to be used and have every input routine use that:

```
istream* input; // pointer to input stream

int main(int argc, char* argv[])
{
    switch (argc) {
    case 1:                                     // read from standard input
        input = &cin;
        break;
    case 2:                                     // read argument string
        input = new istringstream(argv[1]);
        break;
    default:
        error("too many arguments");
        return 1;
    }

    table["pi"] = 3.1415926535897932385;        // insert predefined names
    table["e"] = 2.7182818284590452354;

    while (*input) {
        get_token();
        if (curr_tok == END) break;
        if (curr_tok == PRINT) continue;
        cout << expr(false) << '\n';
    }

    if (input != &cin) delete input;
    return no_of_errors;
}
```

An ***istringstream*** is a kind of ***istream*** that reads from its character string argument (§21.5.3). Upon reaching the end of its string, an ***istringstream*** fails exactly like other streams do when they hit the end of input (§3.6, §21.3.3). To use an ***istringstream***, you must include ***<sstream>***.

It would be easy to modify ***main()*** to accept several command-line arguments, but this does not appear to be necessary, especially as several expressions can be passed as a single argument:

```
dc "rate=1.1934;150/rate;19.75/rate;217/rate"
```

I use quotes because ; is the command separator on my UNIX systems. Other systems have different conventions for supplying arguments to a program on startup.

It was inelegant to modify all of the input routines to use ****input*** rather than ***cin*** to gain the flexibility to use alternative sources of input. The change could have been avoided had I shown foresight by introducing something like ***input*** from the start. A more general and useful view is to note that the source of input really should be the parameter of a calculator module. That is, the fundamental problem with this calculator example is that what I refer to as ''the calculator'' is only a collection of functions and data. There is no module (§2.4) or object (§2.5.2) that explicitly represents the calculator. Had I set out to design a calculator module or a calculator type, I would naturally have considered what its parameters should be (§8.5[3], §10.6[16]).

6.1.8 A Note on Style [expr.style]

To programmers unacquainted with associative arrays, the use of the standard library ***map*** as the symbol table seems almost like cheating. It is not. The standard library and other libraries are meant to be used. Often, a library has received more care in its design and implementation than a programmer could afford for a handcrafted piece of code to be used in just one program.

Looking at the code for the calculator, especially at the first version, we can see that there isn't much traditional C-style, low-level code presented. Many of the traditional tricky details have been replaced by uses of standard library classes such as ***ostream***, ***string***, and ***map*** (§3.4, §3.5, §3.7.4, Chapter 17).

Note the relative scarcity of arithmetic, loops, and even assignments. This is the way things ought to be in code that doesn't manipulate hardware directly or implement low-level abstractions.

6.2 Operator Summary [expr.operators]

This section presents a summary of expressions and some examples. Each operator is followed by one or more names commonly used for it and an example of its use. In these tables, a *class_name* is the name of a class, a *member* is a member name, an *object* is an expression yielding a class object, a *pointer* is an expression yielding a pointer, an *expr* is an expression, and an *lvalue* is an expression denoting a nonconstant object. A *type* can be a fully general type name (with *, (), etc.) only when it appears in parentheses; elsewhere, there are restrictions (§A.5).

The syntax of expressions is independent of operand types. The meanings presented here apply when the operands are of built-in types (§4.1.1). In addition, you can define meanings for operators applied to operands of user-defined types (§2.5.2, Chapter 11).

Operator Summary	
scope resolution	*class_name* `::` *member*
scope resolution	*namespace_name* `::` *member*
global	`::` *name*
global	`::` *qualified-name*
member selection	*object* `.` *member*
member selection	*pointer* `->` *member*
subscripting	*pointer* `[` *expr* `]`
function call	*expr* `(` *expr_list* `)`
value construction	*type* `(` *expr_list* `)`
post increment	*lvalue* `++`
post decrement	*lvalue* `--`
type identification	***typeid*** `(` *type* `)`
run-time type identification	***typeid*** `(` *expr* `)`
run-time checked conversion	***dynamic_cast*** `<` *type* `>(` *expr* `)`
compile-time checked conversion	***static_cast*** `<` *type* `>(` *expr* `)`
unchecked conversion	***reinterpret_cast*** `<` *type* `>(` *expr* `)`
const conversion	***const_cast*** `<` *type* `>(` *expr* `)`
size of object	***sizeof*** *expr*
size of type	***sizeof*** `(` *type* `)`
pre increment	`++` *lvalue*
pre decrement	`--` *lvalue*
complement	`~` *expr*
not	`!` *expr*
unary minus	`-` *expr*
unary plus	`+` *expr*
address of	`&` *lvalue*
dereference	`*` *expr*
create (allocate)	***new*** *type*
create (allocate and initialize)	***new*** *type* `(` *expr-list* `)`
create (place)	***new*** `(` *expr-list* `)` *type*
create (place and initialize)	***new*** `(` *expr-list* `)` *type* `(` *expr-list* `)`
destroy (de-allocate)	***delete*** *pointer*
destroy array	***delete***`[]` *pointer*
cast (type conversion)	`(` *type* `)` *expr*
member selection	*object* `.*` *pointer-to-member*
member selection	*pointer* `->*` *pointer-to-member*
multiply	*expr* `*` *expr*
divide	*expr* `/` *expr*
modulo (remainder)	*expr* `%` *expr*

Operator Summary (continued)	
add (plus)	*expr* + *expr*
subtract (minus)	*expr* - *expr*
shift left	*expr* << *expr*
shift right	*expr* >> *expr*
less than	*expr* < *expr*
less than or equal	*expr* <= *expr*
greater than	*expr* > *expr*
greater than or equal	*expr* >= *expr*
equal	*expr* == *expr*
not equal	*expr* != *expr*
bitwise AND	*expr* & *expr*
bitwise exclusive OR	*expr* ^ *expr*
bitwise inclusive OR	*expr* | *expr*
logical AND	*expr* && *expr*
logical inclusive OR	*expr* || *expr*
simple assignment	*lvalue* = *expr*
multiply and assign	*lvalue* *= *expr*
divide and assign	*lvalue* /= *expr*
modulo and assign	*lvalue* %= *expr*
add and assign	*lvalue* += *expr*
subtract and assign	*lvalue* -= *expr*
shift left and assign	*lvalue* <<= *expr*
shift right and assign	*lvalue* >>= *expr*
AND and assign	*lvalue* &= *expr*
inclusive OR and assign	*lvalue* |= *expr*
exclusive OR and assign	*lvalue* ^= *expr*
conditional expression	*expr* ? *expr* : *expr*
throw exception	**throw** *expr*
comma (sequencing)	*expr* , *expr*

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 +.

Unary operators and assignment operators are right-associative; all others are left-associative. For example, `a=b=c` means `a=(b=c)`, `a+b+c` means `(a+b)+c`, and `*p++` means `*(p++)`, *not* `(*p)++`.

A few grammar rules cannot be expressed in terms of precedence (also known as binding strength) and associativity. For example, `a=b<c?d=e:f=g` means `a=((b<c)?(d=e):(f=g))`, but you need to look at the grammar (§A.5) to determine that.

6.2.1 Results [expr.res]

The result types of arithmetic operators are determined by a set of rules known as ''the usual arithmetic conversions'' (§C.6.3). The overall aim is to produce a result of the ''largest'' operand type. For example, if a binary operator has a floating-point operand, the computation is done using floating-point arithmetic and the result is a floating-point value. If it has a ***long*** operand, the computation is done using long integer arithmetic, and the result is a ***long***. Operands that are smaller than an ***int*** (such as ***bool*** and ***char***) are converted to ***int*** before the operator is applied.

The relational operators, ==, <=, etc., produce Boolean results. The meaning and result type of user-defined operators are determined by their declarations (§11.2).

Where logically feasible, the result of an operator that takes an lvalue operand is an lvalue denoting that lvalue operand. For example:

```
void f(int x, int y)
{
    int j = x = y;          // the value of x=y is the value of x after the assignment
    int* p = &++x;          // p points to x
    int* q = &(x++);        // error: x++ is not an lvalue (it is not the value stored in x)
    int* pp = &(x>y?x:y);   // address of the int with the larger value
}
```

If both the second and third operands of ?: are lvalues and have the same type, the result is of that type and is an lvalue. Preserving lvalues in this way allows greater flexibility in using operators. This is particularly useful when writing code that needs to work uniformly and efficiently with both built-in and user-defined types (e.g., when writing templates or programs that generate C++ code).

The result of ***sizeof*** is of an unsigned integral type called ***size_t*** defined in ***<cstddef>***. The result of pointer subtraction is of a signed integral type called ***ptrdiff_t*** defined in ***<cstddef>***.

Implementations do not have to check for arithmetic overflow and hardly any do. For example:

```
void f()
{
    int i = 1;
    while (0 < i) i++;
    cout << "i has become negative!" << i << '\n';
}
```

This will (eventually) try to increase ***i*** past the largest integer. What happens then is undefined, but typically the value ''wraps around'' to a negative number (on my machine ***-2147483648***). Similarly, the effect of dividing by zero is undefined, but doing so usually causes abrupt termination of the program. In particular, underflow, overflow, and division by zero do not throw standard exceptions (§14.10).

6.2.2 Evaluation Order [expr.evaluation]

The order of evaluation of subexpressions within an expression is undefined. In particular, you cannot assume that the expression is evaluated left to right. For example:

```
int x = f(2)+g(3);    // undefined whether f() or g() is called first
```

Better code can be generated in the absence of restrictions on expression evaluation order. However, the absence of restrictions on evaluation order can lead to undefined results. For example,

```
int i = 1;
v[i] = i++;      // undefined result
```

may be evaluated as either `v[1]=1` or `v[2]=1` or may cause some even stranger behavior. Compilers can warn about such ambiguities. Unfortunately, most do not.

The operators `,` (comma), `&&` (logical and), and `||` (logical or) guarantee that their left-hand operand is evaluated before their right-hand operand. For example, `b=(a=2,a+1)` assigns `3` to `b`. Examples of the use of `||` and `&&` can be found in §6.2.3. For built-in types, the second operand of `&&` is evaluated only if its first operand is ***true***, and the second operand of `||` is evaluated only if its first operand is ***false***; this is sometimes called *short-circuit evaluation*. Note that the sequencing operator `,` (comma) is logically different from the comma used to separate arguments in a function call. Consider:

```
f1(v[i],i++);        // two arguments
f2( (v[i],i++) );    // one argument
```

The call of *f1* has two arguments, `v[i]` and `i++`, and the order of evaluation of the argument expressions is undefined. Order dependence of argument expressions is very poor style and has undefined behavior. The call of *f2* has one argument, the comma expression `(v[i],i++)`, which is equivalent to `i++`.

Parentheses can be used to force grouping. For example, `a*b/c` means `(a*b)/c` so parentheses must be used to get `a*(b/c)`; `a*(b/c)` may be evaluated as `(a*b)/c` only if the user cannot tell the difference. In particular, for many floating-point computations `a*(b/c)` and `(a*b)/c` are significantly different, so a compiler will evaluate such expressions exactly as written.

6.2.3 Operator Precedence [expr.precedence]

Precedence levels and associativity rules reflect the most common usage. For example,

```
if (i<=0 || max<i) // ...
```

means ''if *i* is less than or equal to *0* or if *max* is less than *i*.'' That is, it is equivalent to

```
if ( (i<=0) || (max<i) ) // ...
```

and not the legal but nonsensical

```
if (i <= (0||max) < i) // ...
```

However, parentheses should be used whenever a programmer is in doubt about those rules. Use of parentheses becomes more common as the subexpressions become more complicated, but complicated subexpressions are a source of errors. Therefore, if you start feeling the need for parentheses, you might consider breaking up the expression by using an extra variable.

There are cases when the operator precedence does not result in the ''obvious'' interpretation. For example:

```
if (i&mask == 0)      // oops! == expression as operand for &
```

This does not apply a mask to *i* and then test if the result is zero. Because == has higher precedence than &, the expression is interpreted as *i&(mask==0)*. Fortunately, it is easy enough for a compiler to warn about most such mistakes. In this case, parentheses are important:

```
if ((i&mask) == 0) // ...
```

It is worth noting that the following does not work the way a mathematician might expect:

```
if (0 <= x <= 99) // ...
```

This is legal, but it is interpreted as *(0<=x)<=99*, where the result of the first comparison is either *true* or *false*. This Boolean value is then implicitly converted to *1* or *0*, which is then compared to *99*, yielding *true*. To test whether *x* is in the range *0..99*, we might use:

```
if (0<=x && x<=99) // ...
```

A common mistake for novices is to use = (assignment) instead of == (equals) in a condition:

```
if (a = 7) // oops! constant assignment in condition
```

This is natural because = means ''equals'' in many languages. Again, it is easy for a compiler to warn about most such mistakes – and many do.

6.2.4 Bitwise Logical Operators [expr.logical]

The bitwise logical operators &, |, ^, ~, >>, and << are applied to objects of integer types – that is, *bool*, *char*, *short*, *int*, *long*, and their *unsigned* counterparts. The results are also integers.

A typical use of bitwise logical operators is to implement the notion of a small set (a bit vector). In this case, each bit of an unsigned integer represents one member of the set, and the number of bits limits the number of members. The binary operator & is interpreted as intersection, | as union, ^ as symmetric difference, and ~ as complement. An enumeration can be used to name the members of such a set. Here is a small example borrowed from an implementation of *ostream*:

```
enum ios_base::iostate {
        goodbit=0, eofbit=1, failbit=2, badbit=4
};
```

The implementation of a stream can set and test its state like this:

```
state = goodbit;
// ...
if (state&(badbit|failbit)) // stream no good
```

The extra parentheses are necessary because & has higher precedence than |.

A function that reaches the end of input might report it like this:

```
state |= eofbit;
```

The |= operator is used to add to the state. A simple assignment, *state=eofbit*, would have cleared all other bits.

These stream state flags are observable from outside the stream implementation. For example, we could see how the states of two streams differ like this:

```
int diff = cin.rdstate()^cout.rdstate();    // rdstate() returns the state
```

Computing differences of stream states is not very common. For other similar types, computing differences is essential. For example, consider comparing a bit vector that represents the set of interrupts being handled with another that represents the set of interrupts waiting to be handled.

Please note that this bit fiddling is taken from the implementation of iostreams rather than from the user interface. Convenient bit manipulation can be very important, but for reliability, maintainability, portability, etc., it should be kept at low levels of a system. For more general notions of a set, see the standard library ***set*** (§17.4.3), ***bitset*** (§17.5.3), and ***vector<bool>*** (§16.3.11).

Using fields (§C.8.1) is really a convenient shorthand for shifting and masking to extract bit fields from a word. This can, of course, also be done using the bitwise logical operators. For example, one could extract the middle 16 bits of a 32-bit ***long*** like this:

```
unsigned short middle(long a) { return (a>>8)&0xffff; }
```

Do not confuse the bitwise logical operators with the logical operators: &&, ||, and ! . The latter return either ***true*** or ***false***, and they are primarily useful for writing the test in an ***if***, ***while***, or ***for*** statement (§6.3.2, §6.3.3). For example, !***0*** (not zero) is the value ***true***, whereas ~***0*** (complement of zero) is the bit pattern all-ones, which in two's complement representation is the value -***1***.

6.2.5 Increment and Decrement [expr.incr]

The ++ operator is used to express incrementing directly, rather than expressing it indirectly using a combination of an addition and an assignment. By definition, ++***lvalue*** means ***lvalue+=1***, which again means ***lvalue=lvalue+1*** provided ***lvalue*** has no side effects. The expression denoting the object to be incremented is evaluated once (only). Decrementing is similarly expressed by the -- operator. The operators ++ and -- can be used as both prefix and postfix operators. The value of ++***x*** is the new (that is, incremented) value of ***x***. For example, ***y***=++***x*** is equivalent to ***y***=(***x***+=***1***). The value of ***x***++, however, is the old value of ***x***. For example, ***y***=***x***++ is equivalent to ***y***=(***t***=***x***,***x***+=***1***,***t***), where ***t*** is a variable of the same type as ***x***.

Like addition and subtraction of pointers, ++ and -- on pointers operate in terms of elements of the array into which the pointer points; ***p***++ makes ***p*** point to the next element (§5.3.1).

The increment operators are particularly useful for incrementing and decrementing variables in loops. For example, one can copy a zero-terminated string like this:

```
void cpy(char* p, const char* q)
{
    while (*p++ = *q++) ;
}
```

Like C, C++ is both loved and hated for enabling such terse, expression-oriented coding. Because

```
while (*p++ = *q++) ;
```

is more than a little obscure to non-C programmers and because the style of coding is not uncommon in C and C++, it is worth examining more closely.

Consider first a more traditional way of copying an array of characters:

```
int length = strlen(q);
for (int i = 0; i<=length; i++) p[i] = q[i];
```

This is wasteful. The length of a zero-terminated string is found by reading the string looking for the terminating zero. Thus, we read the string twice: once to find its length and once to copy it. So we try this instead:

```
int i;
for (i = 0; q[i]!=0 ; i++) p[i] = q[i];
p[i] = 0;  // terminating zero
```

The variable **i** used for indexing can be eliminated because **p** and **q** are pointers:

```
while (*q != 0) {
        *p = *q;
        p++;        // point to next character
        q++;        // point to next character
}
*p = 0;             // terminating zero
```

Because the post-increment operation allows us first to use the value and then to increment it, we can rewrite the loop like this:

```
while (*q != 0) {
        *p++ = *q++;
}
*p = 0; // terminating zero
```

The value of `*p++ = *q++` is `*q`. We can therefore rewrite the example like this:

```
while ((*p++ = *q++) != 0) { }
```

In this case, we don't notice that `*q` is zero until we already have copied it into `*p` and incremented **p**. Consequently, we can eliminate the final assignment of the terminating zero. Finally, we can reduce the example further by observing that we don't need the empty block and that the "`!= 0`" is redundant because the result of a pointer or integral condition is always compared to zero anyway. Thus, we get the version we set out to discover:

```
while (*p++ = *q++) ;
```

Is this version less readable than the previous versions? Not to an experienced C or C++ programmer. Is this version more efficient in time or space than the previous versions? Except for the first version that called **strlen()**, not really. Which version is the most efficient will vary among machine architectures and among compilers.

The most efficient way of copying a zero-terminated character string for your particular machine ought to be the standard string copy function:

```
char* strcpy(char*, const char*);    // from <string.h>
```

For more general copying, the standard **copy** algorithm (§2.7.2, §18.6.1) can be used. Whenever possible, use standard library facilities in preference to fiddling with pointers and bytes. Standard library functions may be inlined (§7.1.1) or even implemented using specialized machine

instructions. Therefore, you should measure carefully before believing that some piece of hand-crafted code outperforms library functions.

6.2.6 Free Store [expr.free]

A named object has its lifetime determined by its scope (§4.9.4). However, it is often useful to create an object that exists independently of the scope in which it was created. In particular, it is common to create objects that can be used after returning from the function in which they were created. The operator ***new*** creates such objects, and the operator ***delete*** can be used to destroy them. Objects allocated by ***new*** are said to be ''on the free store'' (also, to be ''heap objects,'' or ''allocated in dynamic memory'').

Consider how we might write a compiler in the style used for the desk calculator (§6.1). The syntax analysis functions might build a tree of the expressions for use by the code generator:

```
struct Enode {
        Token_value oper;
        Enode* left;
        Enode* right;
        // ...
};

Enode* expr(bool get)
{
        Enode* left = term(get);

        for (;;)
                switch(curr_tok) {
                case PLUS:
                case MINUS:
                {       Enode* n = new Enode;     // create an Enode on free store
                        n->oper = curr_tok;
                        n->left = left;
                        n->right = term(true);
                        left = n;
                        break;
                }
                default:
                        return left;              // return node
                }
}
```

A code generator would then use the resulting nodes and delete them:

```
void generate(Enode* n)
{
        switch (n->oper) {
        case PLUS:
                // ...
                delete n;  // delete an Enode from the free store
        }
}
```

An object created by ***new*** exists until it is explicitly destroyed by ***delete***. Then, the space it occupied can be reused by ***new***. A C++ implementation does not guarantee the presence of a ''garbage collector'' that looks out for unreferenced objects and makes them available to ***new*** for reuse. Consequently, I will assume that objects created by ***new*** are manually freed using ***delete***. If a garbage collector is present, the ***delete***s can be omitted in most cases (§C.9.1).

The ***delete*** operator may be applied only to a pointer returned by ***new*** or to zero. Applying ***delete*** to zero has no effect.

More specialized versions of operator ***new*** can also be defined (§15.6).

6.2.6.1 Arrays [expr.array]

Arrays of objects can also be created using ***new***. For example:

```
char* save_string(const char* p)
{
        char* s = new char[strlen(p)+1];
        strcpy(s,p);            // copy from p to s
        return s;
}

int main(int argc, char* argv[])
{
        if (argc < 2) exit(1);
        char* p = save_string(argv[1]);
        // ...
        delete[] p;
}
```

The ''plain'' operator ***delete*** is used to delete individual objects; ***delete***[] is used to delete arrays.

To deallocate space allocated by ***new***, ***delete*** and ***delete***[] must be able to determine the size of the object allocated. This implies that an object allocated using the standard implementation of ***new*** will occupy slightly more space than a static object. Typically, one word is used to hold the object's size.

Note that a ***vector*** (§3.7.1, §16.3) is a proper object and can therefore be allocated and deallocated using plain ***new*** and ***delete***. For example:

```
void f(int n)
{
        vector<int>* p = new vector<int>(n);        // individual object
        int* q = new int[n];                        // array
        // ...
        delete p;
        delete[] q;
}
```

6.2.6.2 Memory Exhaustion [expr.exhaust]

The free store operators ***new***, ***delete***, ***new[]***, and ***delete[]*** are implemented using functions:

```
void* operator new(size_t);      // space for individual object
void operator delete(void*);

void* operator new[](size_t); // space for array
void operator delete[](void*);
```

When operator ***new*** needs to allocate space for an object, it calls ***operator new()*** to allocate a suitable number of bytes. Similarly, when operator ***new*** needs to allocate space for an array, it calls ***operator new[]()***.

The standard implementations of ***operator new()*** and ***operator new[]()*** do not initialize the memory returned.

What happens when ***new*** can find no store to allocate? By default, the allocator throws a ***bad_alloc*** exception. For example:

```
void f()
{
    try {
        for(;;) new char[10000];
    }
    catch(bad_alloc) {
        cerr << "Memory exhausted!\n";
    }
}
```

However much memory we have available, this will eventually invoke the ***bad_alloc*** handler.

We can specify what ***new*** should do upon memory exhaustion. When ***new*** fails, it first calls a function specified by a call to ***set_new_handler()*** declared in ***<new>***, if any. For example:

```
void out_of_store()
{
    cerr << "operator new failed: out of store\n";
    throw bad_alloc();
}

int main()
{
    set_new_handler(out_of_store); // make out_of_store the new_handler
    for (;;) new char[10000];
    cout << "done\n";
}
```

This will never get to write ***done***. Instead, it will write

```
operator new failed: out of store
```

See §14.4.5 for a plausible implementation of an ***operator new()*** that checks to see if there is a new handler to call and that throws ***bad_alloc*** if not. A ***new_handler*** might do something more clever than simply terminating the program. If you know how ***new*** and ***delete*** work – for example,

because you provided your own ***operator new***( ) and ***operator delete***( ) – the handler might attempt to find some memory for ***new*** to return. In other words, a user might provide a garbage collector, thus rendering the use of ***delete*** optional. Doing this is most definitely not a task for a beginner, though. For almost everybody who needs an automatic garbage collector, the right thing to do is to acquire one that has already been written and tested (§C.9.1).

By providing a ***new_handler***, we take care of the check for memory exhaustion for every ordinary use of ***new*** in the program. Two alternative ways of controlling memory allocation exist. We can either provide nonstandard allocation and deallocation functions (§15.6) for the standard uses of ***new*** or rely on additional allocation information provided by the user (§10.4.11, §19.4.5).

6.2.7 Explicit Type Conversion [expr.cast]

Sometimes, we have to deal with‘‘raw memory;’’ that is, memory that holds or will hold objects of a type not known to the compiler. For example, a memory allocator may return a ***void**** pointing to newly allocated memory or we might want to state that a given integer value is to be treated as the address of an I/O device:

```
void* malloc(size_t);

void f()
{
    int* p = static_cast<int*>(malloc(100));          // new allocation used as ints
    IO_device* d1 = reinterpret_cast<IO_device*>(0Xff00);    // device at 0Xff00
    // ...
}
```

A compiler does not know the type of the object pointed to by the ***void****. Nor can it know whether the integer ***0Xff00*** is a valid address. Consequently, the correctness of the conversions are completely in the hands of the programmer. Explicit type conversion, often called *casting*, is occasionally essential. However, traditionally it is seriously overused and a major source of errors.

The ***static_cast*** operator converts between related types such as one pointer type to another, an enumeration to an integral type, or a floating-point type to an integral type. The ***reinterpret_cast*** handles conversions between unrelated types such as an integer to a pointer. This distinction allows the compiler to apply some minimal type checking for ***static_cast*** and makes it easier for a programmer to find the more dangerous conversions represented as ***reinterpret_cast***s. Some ***static_cast***s are portable, but few ***reinterpret_cast***s are. Hardly any guarantees are made for ***reinterpret_cast***, but generally it produces a value of a new type that has the same bit pattern as its argument. If the target has at least as many bits as the original value, we can ***reinterpret_cast*** the result back to its original type and use it. The result of a ***reinterpret_cast*** is guaranteed to be usable only if its result type is the exact type used to define the value involved. Note that ***reinterpret_cast*** is the kind of conversion that must be used for pointers to functions (§7.7).

If you feel tempted to use an explicit type conversion, take the time to consider if it is *really* necessary. In C++, explicit type conversion is unnecessary in most cases when C needs it (§1.6) and also in many cases in which earlier versions of C++ needed it (§1.6.2, §B.2.3). In many programs, explicit type conversion can be completely avoided; in others, its use can be localized to a few routines. In this book, explicit type conversion is used in realistic situations in §6.2.7, §7.7, §13.5, §15.4, and §25.4.1, only.

A form of run-time checked conversion, ***dynamic_cast*** (§15.4.1), and a cast for removing ***const*** qualifiers, ***const_cast*** (§15.4.2.1), are also provided.

From C, C++ inherited the notation `(T)e`, which performs any conversion that can be expressed as a combination of ***static_cast***s, ***reinterpret_cast***s, and ***const_cast***s to make a value of type `T` from the expression `e` (§B.2.3). This C-style cast is far more dangerous than the named conversion operators because the notation is harder to spot in a large program and the kind of conversion intended by the programmer is not explicit. That is, `(T)e` might be doing a portable conversion between related types, a nonportable conversion between unrelated types, or removing the ***const*** modifier from a pointer type. Without knowing the exact types of `T` and `e`, you cannot tell.

6.2.8 Constructors [expr.ctor]

The construction of a value of type `T` from a value `e` can be expressed by the functional notation `T(e)`. For example:

```
void f(double d)
{
    int i = int(d);             // truncate d
    complex z = complex(d);     // make a complex from d
    // ...
}
```

The `T(e)` construct is sometimes referred to as a *function-style cast*. For a built-in type `T`, `T(e)` is equivalent to `static_cast<T>(e)`. Unfortunately, this implies that the use of `T(e)` is not always safe. For arithmetic types, values can be truncated and even explicit conversion of a longer integer type to a shorter (such as ***long*** to ***char***) can result in undefined behavior. I try to use the notation exclusively where the construction of a value is well-defined; that is, for narrowing arithmetic conversions (§C.6), for conversion from integers to enumerations (§4.8), and the construction of objects of user-defined types (§2.5.2, §10.2.3).

Pointer conversions cannot be expressed directly using the `T(e)` notation. For example, `char*(2)` is a syntax error. Unfortunately, the protection that the constructor notation provides against such dangerous conversions can be circumvented by using ***typedef*** names (§4.9.7) for pointer types.

The constructor notation `T()` is used to express the default value of type `T`. For example:

```
void f(double d)
{
    int j = int();              // default int value
    complex z = complex();      // default complex value
    // ...
}
```

The value of an explicit use of the constructor for a built-in type is `0` converted to that type (§4.9.5). Thus, `int()` is another way of writing `0`. For a user-defined type `T`, `T()` is defined by the default constructor (§10.4.2), if any.

The use of the constructor notation for built-in types is particularly important when writing templates. Then, the programmer does not know whether a template parameter will refer to a built-in type or a user-defined type (§16.3.4, §17.4.1.2).

6.3 Statement Summary [expr.stmts]

Here are a summary and some examples of C++ statements:

Statement Syntax

statement:
    declaration
    { *statement-list*$_{opt}$ }
    try { *statement-list*$_{opt}$ } *handler-list*
    expression$_{opt}$;

    if (*condition*) *statement*
    if (*condition*) *statement* ***else*** *statement*
    switch (*condition*) *statement*

    while (*condition*) *statement*
    do *statement* ***while*** (*expression*) ;
    for (*for-init-statement condition*$_{opt}$; *expression*$_{opt}$) *statement*

    case *constant-expression* : *statement*
    default : *statement*
    break ;
    continue ;

    return *expression*$_{opt}$;

    goto *identifier* ;
    identifier : *statement*

statement-list:
    statement statement-list$_{opt}$

condition:
    expression
    type-specifier declarator = *expression*

handler-list:
    catch (*exception-declaration*) { *statement-list*$_{opt}$ }
    handler-list handler-list$_{opt}$

Note that a declaration is a statement and that there is no assignment statement or procedure call statement; assignments and function calls are expressions. The statements for handling exceptions, *try-block*s, are described in §8.3.1.

6.3.1 Declarations as Statements [expr.dcl]

A declaration is a statement. Unless a variable is declared ***static***, its initializer is executed whenever the thread of control passes through the declaration (see also §10.4.8). The reason for allowing declarations wherever a statement can be used (and a few other places; §6.3.2.1, §6.3.3.1) is to enable the programmer to minimize the errors caused by uninitialized variables and to allow better locality in code. There is rarely a reason to introduce a variable before there is a value for it to hold. For example:

```
void f(vector<string>& v, int i, const char* p)
{
    if (p==0) return;
    if (i<0 || v.size()<=i) error("bad index");
    string s = v[i];
    if (s == p) {
        // ...
    }
    // ...
}
```

The ability to place declarations after executable code is essential for many constants and for single-assignment styles of programming where a value of an object is not changed after initialization. For user-defined types, postponing the definition of a variable until a suitable initializer is available can also lead to better performance. For example,

```
string s; /* ... */ s = "The best is the enemy of the good.";
```

can easily be much slower than

```
string s = "Voltaire";
```

The most common reason to declare a variable without an initializer is that it requires a statement to initialize it. Examples are input variables and arrays.

6.3.2 Selection Statements [expr.select]

A value can be tested by either an ***if*** statement or a ***switch*** statement:

```
if ( condition ) statement
if ( condition ) statement else statement
switch ( condition ) statement
```

The comparison operators

```
==   !=   <   <=   >   >=
```

return the ***bool*** ***true*** if the comparison is true and ***false*** otherwise.

In an ***if*** statement, the first (or only) statement is executed if the expression is nonzero and the second statement (if it is specified) is executed otherwise. This implies that any arithmetic or pointer expression can be used as a condition. For example, if ***x*** is an integer, then

```
if (x) // ...
```

means

```
if (x != 0) // ...
```

For a pointer **p**,

```
if (p) // ...
```

is a direct statement of the test ''does **p** point to a valid object,'' whereas

```
if (p != 0) // ...
```

states the same question indirectly by comparing to a value known not to point to an object. Note that the representation of the pointer **0** is not all-zeros on all machines (§5.1.1). Every compiler I have checked generated the same code for both forms of the test.

The logical operators

```
&&   ||   !
```

are most commonly used in conditions. The operators && and || will not evaluate their second argument unless doing so is necessary. For example,

```
if (p && 1<p->count) // ...
```

first tests that **p** is nonzero. It tests **1<p->count** only if **p** is nonzero.

Some *if-statement*s can conveniently be replaced by *conditional-expression*s. For example,

```
if (a <= b)
        max = b;
else
        max = a;
```

is better expressed like this:

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

The parentheses around the condition are not necessary, but I find the code easier to read when they are used.

A *switch-statement* can alternatively be written as a set of ***if-statement***s. For example,

```
switch (val) {
case 1:
        f();
        break;
case 2:
        g();
        break;
default:
        h();
        break;
}
```

could alternatively be expressed as

```
if (val == 1)
        f();
else if (val == 2)
        g();
else
        h();
```

The meaning is the same, but the first (***switch***) version is preferred because the nature of the operation (testing a value against a set of constants) is explicit. This makes the ***switch*** statement easier to read for nontrivial examples. It can also lead to the generation of better code.

Beware that a case of a switch must be terminated somehow unless you want to carry on executing the next case. Consider:

```
switch (val) {                  // beware
case 1:
        cout << "case 1\n";
case 2:
        cout << "case 2\n";
default:
        cout << "default: case not found\n";
}
```

Invoked with ***val==1***, this prints

```
case 1
case 2
default: case not found
```

to the great surprise of the uninitiated. It is a good idea to comment the (rare) cases in which a fall-through is intentional so that an uncommented fall-through can be assumed to be an error. A ***break*** is the most common way of terminating a case, but a ***return*** is often useful (§6.1.1).

6.3.2.1 Declarations in Conditions [expr.cond]

To avoid accidental misuse of a variable, it is usually a good idea to introduce the variable into the smallest scope possible. In particular, it is usually best to delay the definition of a local variable until one can give it an initial value. That way, one cannot get into trouble by using the variable before its initial value is assigned.

One of the most elegant applications of these two principles is to declare a variable in a condition. Consider:

```
if (double d = prim(true)) {
        left /= d;
        break;
}
```

Here, ***d*** is declared and initialized and the value of ***d*** after initialization is tested as the value of the condition. The scope of ***d*** extends from its point of declaration to the end of the statement that the condition controls. For example, had there been an ***else***-branch to the *if-statement*, ***d*** would be in scope on both branches.

The obvious and traditional alternative is to declare ***d*** before the condition. However, this opens the scope (literally) for the use of ***d*** before its initialization or after its intended useful life:

```
double d;
// ...

d2 = d;     // oops!
// ...

if (d = prim(true)) {
        left /= d;
        break;
}
// ...

d = 2.0; // two unrelated uses of d
```

In addition to the logical benefits of declaring variables in conditions, doing so also yields the most compact source code.

A declaration in a condition must declare and initialize a single variable or ***const***.

6.3.3 Iteration Statements [expr.loop]

A loop can be expressed as a ***for***, ***while***, or ***do*** statement:

```
while ( condition ) statement
do statement while ( expression ) ;
for ( for-init-statement condition_opt ; expression_opt ) statement
```

Each of these statements executes a statement (called the *controlled* statement or the *body of the loop*) repeatedly until the condition becomes false or the programmer breaks out of the loop some other way.

The *for-statement* is intended for expressing fairly regular loops. The loop variable, the termination condition, and the expression that updates the loop variable can be presented ''up front'' on a single line. This can greatly increase readability and thereby decrease the frequency of errors. If no initialization is needed, the initializing statement can be empty. If the *condition* is omitted, the *for-statement* will loop forever unless the user explicitly exits it by a ***break***, ***return***, ***goto***, ***throw***, or some less obvious way such as a call of ***exit***( ) (§9.4.1.1). If the *expression* is omitted, we must update some form of loop variable in the body of the loop. If the loop isn't of the simple ''introduce a loop variable, test the condition, update the loop variable'' variety, it is often better expressed as a *while-statement*. A *for-statement* is also useful for expressing a loop without an explicit termination condition:

```
for(;;) { // ''forever''
        // ...
}
```

A *while-statement* simply executes its controlled statement until its condition becomes ***false***. I tend to prefer *while-statement*s over *for-statement*s when there isn't an obvious loop variable or where the update of a loop variable naturally comes in the middle of the loop body. An input loop is an example of a loop where there is no obvious loop variable:

```
while(cin>>ch) // ...
```

In my experience, the *do-statement* is a source of errors and confusion. The reason is that its body is always executed once before the condition is evaluated. However, for the body to work correctly, something very much like the condition must hold even the first time through. More often than I would have guessed, I have found that condition not to hold as expected either when the program was first written and tested or later after the code preceding it has been modified. I also prefer the condition ''up front where I can see it.'' Consequently, I tend to avoid *do-statement*s.

6.3.3.1 Declarations in For-Statements [expr.for]

A variable can be declared in the initializer part of a *for-statement*. If that initializer is a declaration, the variable (or variables) it introduces is in scope until the end of the *for-statement*. For example:

```
void f(int v[], int max)
{
    for (int i = 0; i<max; i++) v[i] = i*i;
}
```

If the final value of an index needs to be known after exit from a ***for***-loop, the index variable must be declared outside the ***for***-loop (e.g., §6.3.4).

6.3.4 Goto [expr.goto]

C++ possesses the infamous ***goto***:

```
goto identifier ;
identifier : statement
```

The ***goto*** has few uses in general high-level programming, but it can be very useful when C++ code is generated by a program rather than written directly by a person; for example, ***goto***s can be used in a parser generated from a grammar by a parser generator. The ***goto*** can also be important in the rare cases in which optimal efficiency is essential, for example, in the inner loop of some real-time application.

One of the few sensible uses of ***goto*** in ordinary code is to break out from a nested loop or *switch-statement* (a ***break*** breaks out of only the innermost enclosing loop or *switch-statement*). For example:

```
void f()
{
    int i;
    int j;
```

```
        for (i = 0; i<n; i++)
                for (j = 0; j<m; j++) if (nm[i][j] == a) goto found;
        // not found
        // ...
found:
        // nm[i][j] == a
}
```

There is also a ***continue*** statement that, in effect, goes to the end of a loop statement, as explained in §6.1.5.

6.4 Comments and Indentation [expr.comment]

Judicious use of comments and consistent use of indentation can make the task of reading and understanding a program much more pleasant. Several different consistent styles of indentation are in use. I see no fundamental reason to prefer one over another (although, like most programmers, I have my preferences, and this book reflects them). The same applies to styles of comments.

Comments can be misused in ways that seriously affect the readability of a program. The compiler does not understand the contents of a comment, so it has no way of ensuring that a comment

[1] is meaningful,
[2] describes the program, and
[3] is up to date.

Most programs contain comments that are incomprehensible, ambiguous, and just plain wrong. Bad comments can be worse than no comments.

If something can be stated *in the language itself*, it should be, and not just mentioned in a comment. This remark is aimed at comments such as these:

```
// variable "v" must be initialized

// variable "v" must be used only by function "f()"

// call function "init()" before calling any other function in this file

// call function "cleanup()" at the end of your program

// don't use function "weird()"

// function "f()" takes two arguments
```

Such comments can often be rendered unnecessary by proper use of C++. For example, one might utilize the linkage rules (§9.2) and the visibility, initialization, and cleanup rules for classes (see §10.4.1) to make the preceding examples redundant.

Once something has been stated clearly in the language, it should not be mentioned a second time in a comment. For example:

```
a = b+c;    // a becomes b+c
count++;    // increment the counter
```

Such comments are worse than simply redundant. They increase the amount of text the reader has to look at, they often obscure the structure of the program, and they may be wrong. Note, however,

that such comments are used extensively for teaching purposes in programming language textbooks such as this. This is one of the many ways a program in a textbook differs from a real program.

My preference is for:

[1] A comment for each source file stating what the declarations in it have in common, references to manuals, general hints for maintenance, etc.
[2] A comment for each class, template, and namespace
[3] A comment for each nontrivial function stating its purpose, the algorithm used (unless it is obvious), and maybe something about the assumptions it makes about its environment
[4] A comment for each global and namespace variable and constant
[5] A few comments where the code is nonobvious and/or nonportable
[6] Very little else

For example:

```
//    tbl.c: Implementation of the symbol table.

/*
      Gaussian elimination with partial pivoting.
      See Ralston: "A first course ..." pg 411.
*/

//    swap() assumes the stack layout of an SGI R6000.

/*************************************

      Copyright (c) 1997 AT&T, Inc.
      All rights reserved

*************************************/
```

A well-chosen and well-written set of comments is an essential part of a good program. Writing good comments can be as difficult as writing the program itself. It is an art well worth cultivating.

Note also that if `//` comments are used exclusively in a function, then any part of that function can be commented out using `/* */` style comments, and vice versa.

6.5 Advice [expr.advice]

[1] Prefer the standard library to other libraries and to ‘‘handcrafted code;’’ §6.1.8.
[2] Avoid complicated expressions; §6.2.3.
[3] If in doubt about operator precedence, parenthesize; §6.2.3.
[4] Avoid explicit type conversion (casts); §6.2.7.
[5] When explicit type conversion is necessary, prefer the more specific cast operators to the C-style cast; §6.2.7.
[6] Use the `T(e)` notation exclusively for well-defined construction; §6.2.8.
[7] Avoid expressions with undefined order of evaluation; §6.2.2.
[8] Avoid *goto*; §6.3.4.
[9] Avoid *do-statement*s; §6.3.3.
[10] Don’t declare a variable until you have a value to initialize it with; §6.3.1, §6.3.2.1, §6.3.3.1.

[11] Keep comments crisp; §6.4.
[12] Maintain a consistent indentation style; §6.4.
[13] Prefer defining a member *operator new*() (§15.6) to replacing the global *operator new*(); §6.2.6.2.
[14] When reading input, always consider ill-formed input; §6.1.3.

6.6 Exercises [expr.exercises]

1. (∗1) Rewrite the following ***for*** statement as an equivalent ***while*** statement:

```
for (i=0; i<max_length; i++) if (input_line[i] == '?') quest_count++;
```

   Rewrite it to use a pointer as the controlled variable, that is, so that the test is of the form `*p=='?'`.
2. (∗1) Fully parenthesize the following expressions:

```
a = b + c * d << 2 & 8
a & 077 != 3
a == b || a == c && c < 5
c = x != 0
0 <= i < 7
f(1,2)+3
a = - 1 + + b - - - 5
a = b == c ++
a = b = c = 0
a[4][2] *= * b ? c : * d * 2
a-b,c=d
```

3. (∗2) Read a sequence of possibly whitespace-separated (name,value) pairs, where the name is a single whitespace-separated word and the value is an integer or a floating-point value. Compute and print the sum and mean for each name and the sum and mean for all names. Hint: §6.1.8.
4. (∗1) Write a table of values for the bitwise logical operations (§6.2.4) for all possible combinations of ***0*** and ***1*** operands.
5. (∗1.5) Find 5 different C++ constructs for which the meaning is undefined (§C.2). (∗1.5) Find 5 different C++ constructs for which the meaning is implementation-defined (§C.2).
6. (∗1) Find 10 different examples of nonportable C++ code.
7. (∗2) Write 5 expressions for which the order of evaluation is undefined. Execute them to see what one or – preferably – more implementations do with them.
8. (∗1.5) What happens if you divide by zero on your system? What happens in case of overflow and underflow?
9. (∗1) Fully parenthesize the following expressions:

```
*p++
*--p
++a--
(int*)p->m
*p.m
*a[i]
```

10. (*2) Write these functions: ***strlen***( ), which returns the length of a C-style string; ***strcpy***( ), which copies a string into another; and ***strcmp***( ), which compares two strings. Consider what the argument types and return types ought to be. Then compare your functions with the standard library versions as declared in <***cstring***> (<***string.h***>) and as specified in §20.4.1.
11. (∗1) See how your compiler reacts to these errors:

```
void f(int a, int b)
{
    if (a = 3) // ...
    if (a&077 == 0) // ...
    a := b+1;
}
```

Devise more simple errors and see how the compiler reacts.

12. (∗2) Modify the program from §6.6[3] to also compute the median.
13. (∗2) Write a function ***cat***( ) that takes two C-style string arguments and returns a string that is the concatenation of the arguments. Use ***new*** to find store for the result.
14. (∗2) Write a function ***rev***( ) that takes a string argument and reverses the characters in it. That is, after ***rev***(***p***) the last character of ***p*** will be the first, etc.
15. (∗1.5) What does the following example do?

```
void send(int* to, int* from, int count)
//    Duff's device. Helpful comment deliberately deleted.
{
    int n = (count+7)/8;
    switch (count%8) {
    case 0:    do { *to++ = *from++;
    case 7:         *to++ = *from++;
    case 6:         *to++ = *from++;
    case 5:         *to++ = *from++;
    case 4:         *to++ = *from++;
    case 3:         *to++ = *from++;
    case 2:         *to++ = *from++;
    case 1:         *to++ = *from++;
            } while (--n>0);
    }
}
```

Why would anyone write something like that?

16. (∗2) Write a function ***atoi***(***const char****) that takes a string containing digits and returns the corresponding ***int***. For example, ***atoi***("***123***") is ***123***. Modify ***atoi***( ) to handle C++ octal and hexadecimal notation in addition to plain decimal numbers. Modify ***atoi***( ) to handle the C++

character constant notation.

17. (∗2) Write a function ***itoa(int i, char b[])*** that creates a string representation of ***i*** in ***b*** and returns ***b***.
18. (*2) Type in the calculator example and get it to work. Do not ''save time'' by using an already entered text. You'll learn most from finding and correcting ''little silly errors.''
19. (∗2) Modify the calculator to report line numbers for errors.
20. (∗3) Allow a user to define functions in the calculator. Hint: Define a function as a sequence of operations just as a user would have typed them. Such a sequence can be stored either as a character string or as a list of tokens. Then read and execute those operations when the function is called. If you want a user-defined function to take arguments, you will have to invent a notation for that.
21. (∗1.5) Convert the desk calculator to use a ***symbol*** structure instead of using the static variables ***number_value*** and ***string_value***.
22. (∗2.5) Write a program that strips comments out of a C++ program. That is, read from ***cin***, remove both `//` comments and `/* */` comments, and write the result to ***cout***. Do not worry about making the layout of the output look nice (that would be another, and much harder, exercise). Do not worry about incorrect programs. Beware of `//`, `/*`, and `*/` in comments, strings, and character constants.
23. (∗2) Look at some programs to get an idea of the variety of indentation, naming, and commenting styles actually used.