11

Operator Overloading

*When **I** use a word it means just what*
I choose it to mean – neither more nor less.
– Humpty Dumpty

Notation — operator functions — binary and unary operators — predefined meanings for operators — user-defined meanings for operators — operators and namespaces — a complex type — member and nonmember operators — mixed-mode arithmetic — initialization — copying — conversions — literals — helper functions — conversion operators — ambiguity resolution — friends — members and friends — large objects — assignment and initialization — subscripting — function call — dereferencing — increment and decrement — a string class — advice — exercises.

11.1 Introduction [over.intro]

Every technical field – and most nontechnical fields – have developed conventional shorthand notation to make convenient the presentation and discussion involving frequently-used concepts. For example, because of long acquaintance

```
x+y*z
```

is clearer to us than

```
multiply y by z and add the result to x
```

It is hard to overestimate the importance of concise notation for common operations.

Like most languages, C++ supports a set of operators for its built-in types. However, most concepts for which operators are conventionally used are not built-in types in C++, so they must be represented as user-defined types. For example, if you need complex arithmetic, matrix algebra, logic signals, or character strings in C++, you use classes to represent these notions. Defining operators

The C++ Programming Language, Third Edition by Bjarne Stroustrup. Copyright ©1997 by AT&T.
Published by Addison Wesley Longman, Inc. ISBN 0-201-88954-4. All rights reserved.

for such classes sometimes allows a programmer to provide a more conventional and convenient notation for manipulating objects than could be achieved using only the basic functional notation. For example,

```
class complex {               // very simplified complex
     double re, im;
public:
     complex(double r, double i) : re(r), im(i) { }
     complex operator+(complex);
     complex operator*(complex);
};
```

defines a simple implementation of the concept of complex numbers. A *complex* is represented by a pair of double-precision floating-point numbers manipulated by the operators + and *. The programmer defines *complex*::*operator*+() and *complex*::*operator**() to provide meanings for + and *, respectively. For example, if *b* and *c* are of type *complex*, *b*+*c* means *b*.*operator*+(*c*). We can now approximate the conventional interpretation of *complex* expressions:

```
void f()
{
     complex a = complex(1, 3.1);
     complex b = complex(1.2, 2);
     complex c = b;

     a = b+c;
     b = b+c*a;
     c = a*b+complex(1,2);
}
```

The usual precedence rules hold, so the second statement means *b*=*b*+(*c***a*), not *b*=(*b*+*c*)**a*.

Many of the most obvious uses of operator overloading are for concrete types (§10.3). However, the usefulness of user-defined operators is not restricted to concrete types. For example, the design of general and abstract interfaces often leads to the use of operators such as ->, [], and ().

11.2 Operator Functions [over.oper]

Functions defining meanings for the following operators (§6.2) can be declared:

```
+      -      *      /      %      ^      &
|      ~      !      =      <      >      +=
-=     *=     /=     %=     ^=     &=     |=
<<     >>     >>=    <<=    ==     !=     <=
>=     &&     ||     ++     --     ->*    ,
->     []     ()     new    new[]  delete delete[]
```

The following operators cannot be defined by a user:

:: (scope resolution; §4.9.4, §10.2.4),
. (member selection; §5.7), and
.* (member selection through pointer to function; §15.5).

They take a name, rather than a value, as their second operand and provide the primary means of referring to members. Allowing them to be overloaded would lead to subtleties [Stroustrup,1994].

It is not possible to define new operator tokens, but you can use the function-call notation when this set of operators is not adequate. For example, use *pow*(), not **. These restrictions may seem Draconian, but more flexible rules can easily lead to ambiguities. For example, defining an operator ** to mean exponentiation may seem an obvious and easy task at first glance, but think again. Should ** bind to the left (as in Fortran) or to the right (as in Algol)? Should the expression *a****p* be interpreted as *a**(**p*) or as (*a*)**(*p*)?

The name of an operator function is the keyword *operator* followed by the operator itself; for example, *operator*<<. An operator function is declared and can be called like any other function. A use of the operator is only a shorthand for an explicit call of the operator function. For example:

```
void f(complex a, complex b)
{
        complex c = a + b;                // shorthand
        complex d = a.operator+(b);       // explicit call
}
```

Given the previous definition of *complex*, the two initializers are synonymous.

11.2.1 Binary and Unary Operators [over.binary]

A binary operator can be defined by either a nonstatic member function taking one argument or a nonmember function taking two arguments. For any binary operator @, *aa*@*bb* can be interpreted as either *aa*.*operator*@(*bb*) or *operator*@(*aa*,*bb*). If both are defined, overload resolution (§7.4) determines which, if any, interpretation is used. For example:

```
class X {
public:
        void operator+(int);
        X(int);
};

void operator+(X,X);
void operator+(X,double);

void f(X a)
{
        a+1;        // a.operator+(1)
        1+a;        // ::operator+(X(1),a)
        a+1.0;      // ::operator+(a,1.0)
}
```

A unary operator, whether prefix or postfix, can be defined by either a nonstatic member function taking no arguments or a nonmember function taking one argument. For any prefix unary operator @, @*aa* can be interpreted as either *aa*.*operator*@() or *operator*@(*aa*). If both are defined, overload resolution (§7.4) determines which, if any, interpretation is used. For any postfix unary operator @, *aa*@ can be interpreted as either *aa*.*operator*@(*int*) or *operator*@(*aa*,*int*). This is explained further in §11.11. If both are defined, overload resolution (§7.4) determines which, if any, interpretation is used. An operator can be declared only for the syntax defined for it in the grammar (§A.5). For example, a user cannot define a unary % or a ternary +. Consider:

```
class X {
        // members (with implicit 'this' pointer):

        X* operator&();          // prefix unary & (address of)
        X operator&(X);          // binary & (and)
        X operator++(int);       // postfix increment (see §11.11)
        X operator&(X,X);        // error: ternary
        X operator/();           // error: unary /
};

// nonmember functions :

X operator-(X);                  // prefix unary minus
X operator-(X,X);                // binary minus
X operator--(X&,int);            // postfix decrement
X operator-();                   // error: no operand
X operator-(X,X,X);              // error: ternary
X operator%(X);                  // error: unary %
```

Operator [] is described in §11.8, operator () in §11.9, operator -> in §11.10, operators ++ and -- in §11.11, and the allocation and deallocation operators in §6.2.6.2, §10.4.11, and §15.6.

11.2.2 Predefined Meanings for Operators [over.predefined]

Only a few assumptions are made about the meaning of a user-defined operator. In particular, *operator*=, *operator*[], *operator*(), and *operator*-> must be nonstatic member functions; this ensures that their first operands will be lvalues (§4.9.6).

The meanings of some built-in operators are defined to be equivalent to some combination of other operators on the same arguments. For example, if *a* is an int, *++a* means *a+=1*, which in turn means *a=a+1*. Such relations do not hold for user-defined operators unless the user happens to define them that way. For example, a compiler will not generate a definition of *Z::operator+=()* from the definitions of *Z::operator+()* and *Z::operator=()*.

Because of historical accident, the operators = (assignment), & (address-of), and , (sequencing; §6.2.2) have predefined meanings when applied to class objects. These predefined meanings can be made inaccessible to general users by making them private:

```
class X {
private:
        void operator=(const X&);
        void operator&();
        void operator,(const X&);
        // ...
};
```

```
void f(X a, X b)
{
     a = b;       // error: operator= private
     &a;          // error: operator& private
     a,b;         // error: operator, private
}
```

Alternatively, they can be given new meanings by suitable definitions.

11.2.3 Operators and User-Defined Types [over.user]

An operator function must either be a member or take at least one argument of a user-defined type (functions redefining the *new* and *delete* operators need not). This rule ensures that a user cannot change the meaning of an expression unless the expression contains an object of a user-defined type. In particular, it is not possible to define an operator function that operates exclusively on pointers. This ensures that C++ is extensible but not mutable (with the exception of operators =, &, and , for class objects).

An operator function intended to accept a basic type as its first operand cannot be a member function. For example, consider adding a complex variable *aa* to the integer *2*: *aa+2* can, with a suitably declared member function, be interpreted as *aa.operator+(2)*, but *2+aa* cannot because there is no class *int* for which to define + to mean *2.operator+(aa)*. Even if there were, two different member functions would be needed to cope with *2+aa* and *aa+2*. Because the compiler does not know the meaning of a user-defined +, it cannot assume that it is commutative and so interpret *2+aa* as *aa+2*. This example is trivially handled using nonmember functions (§11.3.2, §11.5).

Enumerations are user-defined types so that we can define operators for them. For example:

```
enum Day { sun, mon, tue, wed, thu, fri, sat };

Day& operator++(Day& d)
{
     return d = (sat==d) ? sun : Day(d+1);
}
```

Every expression is checked for ambiguities. Where a user-defined operator provides a possible interpretation, the expression is checked according to the rules in §7.4.

11.2.4 Operators in Namespaces [over.namespace]

An operator is either a member of a class or defined in some namespace (possibly the global namespace). Consider this simplified version of string I/O from the standard library:

```
namespace std {              // simplified std

     class ostream {
          // ...
          ostream& operator<<(const char*);
     };

     extern ostream cout;
```

```
          class string {
               // ...
          };

          ostream& operator<<(ostream&, const string&);
     }

     int main()
     {
          char* p = "Hello";
          std::string s = "world";
          std::cout << p << ", " << s << "!\n";
     }
```

Naturally, this writes out `Hello, world!` But why? Note that I didn't make everything from `std` accessible by writing:

```
     using namespace std;
```

Instead, I used the `std::` prefix for `string` and `cout`. In other words, I was at my best behavior and didn't pollute the global namespace or in other ways introduce unnecessary dependencies.

The output operator for C-style strings (`char*`) is a member of `std::ostream`, so by definition

```
     std::cout << p
```

means

```
     std::cout.operator<<(p)
```

However, `std::ostream` doesn't have a member function to output a `std::string`, so

```
     std::cout << s
```

means

```
     operator<<(std::cout,s)
```

Operators defined in namespaces can be found based on their operand types just like functions can be found based on their argument types (§8.2.6). In particular, `cout` is in namespace `std`, so `std` is considered when looking for a suitable definition of `<<`. In that way, the compiler finds and uses:

```
     std::operator<<(std::ostream&, const std::string&)
```

For a binary operator `@`, `x@y` where `x` is of type `X` and `y` is of type `Y` is resolved like this:

[1] If `X` is a class, determine whether class `X` or a base of `X` defines `operator@` as a member; if so, that is the `@` to try to use.

[2] Otherwise,
- look for declarations of `@` in the context surrounding `x@y`; and
- if `X` is defined in namespace `N`, look for declarations of `@` in `N`; and
- if `Y` is defined in namespace `M`, look for declarations of `@` in `M`.

If declarations of `operator@` are found in the surrounding context, in `N`, or in `M`, we try to use those operators.

In either case, declarations for several `operator@`s may be found and overload resolution rules

(§7.4) are used to find the best match, if any. This lookup mechanism is applied only if the operator has at least one operand of a user-defined type. Therefore, user-defined conversions (§11.3.2, §11.4) will be considered. Note that a *typedef* name is just a synonym and not a user-defined type (§4.9.7).

11.3 A Complex Number Type [over.complex]

The implementation of complex numbers presented in the introduction is too restrictive to please anyone. For example, from looking at a math textbook we would expect this to work:

```
void f()
{
    complex a = complex(1,2);
    complex b = 3;
    complex c = a+2.3;
    complex d = 2+b;
    complex e = -b-c;
    b = c*2*c;
}
```

In addition, we would expect to be provided with a few additional operators, such as `==` for comparison and `<<` for output, and a suitable set of mathematical functions, such as *sin*`()` and *sqrt*`()`.

Class *complex* is a concrete type, so its design follows the guidelines from §10.3. In addition, users of complex arithmetic rely so heavily on operators that the definition of *complex* brings into play most of the basic rules for operator overloading.

11.3.1 Member and Nonmember Operators [over.member]

I prefer to minimize the number of functions that directly manipulate the representation of an object. This can be achieved by defining only operators that inherently modify the value of their first argument, such as `+=`, in the class itself. Operators that simply produce a new value based on the values of its arguments, such as `+`, are then defined outside the class and use the essential operators in their implementation:

```
class complex {
    double re, im;
public:
    complex& operator+=(complex a);    // needs access to representation
    // ...
};

complex operator+(complex a, complex b)
{
    complex r = a;
    return r += b;  // access representation through +=
}
```

Given these declarations, we can write:

```
void f(complex x, complex y, complex z)
{
        complex r1 = x+y+z;  // r1 = operator+(x,operator+(y,z))
        complex r2 = x;      // r2 = x
        r2 += y;             // r2.operator+=(y)
        r2 += z;             // r2.operator+=(z)
}
```

Except for possible efficiency differences, the computations of *r1* and *r2* are equivalent.

Composite assignment operators such as += and *= tend to be simpler to define than their ‘‘simple’’ counterparts + and *. This surprises most people at first, but it follows from the fact that three objects are involved in a + operation (the two operands and the result), whereas only two objects are involved in a += operation. In the latter case, run-time efficiency is improved by eliminating the need for temporary variables. For example:

```
inline complex& complex::operator+=(complex a)
{
        re += a.re;
        im += a.im;
        return *this;
}
```

does not require a temporary variable to hold the result of the addition and is simple for a compiler to inline perfectly.

A good optimizer will generate close to optimal code for uses of the plain + operator also. However, we don’t always have a good optimizer and not all types are as simple as *complex*, so §11.5 discusses ways of defining operators with direct access to the representation of classes.

11.3.2 Mixed-Mode Arithmetic [over.mixed]

To cope with

```
complex d = 2+b;
```

we need to define operator + to accept operands of different types. In Fortran terminology, we need *mixed-mode arithmetic*. We can achieve that simply by adding appropriate versions of the operators:

```
class complex {
        double re, im;

public:
        complex& operator+=(complex a) {
                re += a.re;
                im += a.im;
                return *this;
        }
```

```
          complex& operator+=(double a) {
               re += a;
               return *this;
          }

          // ...
     };

     complex operator+(complex a, complex b)
     {
          complex r = a;
          return r += b;  // calls complex::operator+=(complex)
     }

     complex operator+(complex a, double b)
     {
          complex r = a;
          return r += b;  // calls complex::operator+=(double)
     }

     complex operator+(double a, complex b)
     {
          complex r = b;
          return r += a;  // calls complex::operator+=(double)
     }
```

Adding a ***double*** to a complex number is a simpler operation than adding a ***complex***. This is reflected in these definitions. The operations taking ***double*** operands do not touch the imaginary part of a complex number and thus will be more efficient.

Given these declarations, we can write:

```
     void f(complex x, complex y)
     {
          complex r1 = x+y;     // calls operator+(complex,complex)
          complex r2 = x+2;     // calls operator+(complex,double)
          complex r3 = 2+x;     // calls operator+(double,complex)
     }
```

11.3.3 Initialization [over.ctor]

To cope with assignments and initialization of ***complex*** variables with scalars, we need a conversion of a scalar (integer or floating-point number) to a ***complex***. For example:

```
     complex b = 3; // should mean b.re=3, b.im=0
```

A constructor taking a single argument specifies a conversion from its argument type to the constructor's type. For example:

```
class complex {
      double re, im;
public:
      complex(double r) :re(r), im(0) { }
      // ...
};
```

The constructor specifies the traditional embedding of the real line in the complex plane.

A constructor is a prescription for creating a value of a given type. The constructor is used when a value of a type is expected and when such a value can be created by a constructor from the value supplied as an initializer or assigned value. Thus, a constructor requiring a single argument need not be called explicitly. For example,

```
complex b = 3;
```

means

```
complex b = complex(3);
```

A user-defined conversion is implicitly applied only if it is unique (§7.4). See §11.7.1 for a way of specifying constructors that can only be explicitly invoked.

Naturally, we still need the constructor that takes two doubles, and a default constructor initializing a *complex* to *(0,0)* is also useful:

```
class complex {
      double re, im;
public:
      complex() : re(0), im(0) { }
      complex(double r) : re(r), im(0) { }
      complex(double r, double i) : re(r), im(i) { }
      // ...
};
```

Using default arguments, we can abbreviate:

```
class complex {
      double re, im;
public:
      complex(double r =0, double i =0) : re(r), im(i) { }
      // ...
};
```

When a constructor is explicitly declared for a type, it is not possible to use an initializer list (§5.7, §4.9.5) as the initializer. For example:

```
complex z1 = { 3 };         // error: complex has a constructor
complex z2 = { 3, 4 };      // error: complex has a constructor
```

11.3.4 Copying [over.copy]

In addition to the explicitly declared constructors, *complex* by default gets a copy constructor defined (§10.2.5). A default copy constructor simply copies all members. To be explicit, we could equivalently have written:

```
class complex {
    double re, im;
public:
    complex(const complex& c) : re(c.re), im(c.im) { }
    // ...
};
```

However, for types where the default copy constructor has the right semantics, I prefer to rely on that default. It is less verbose than anything I can write, and people should understand the default. Also, compilers know about the default and its possible optimization opportunities. Furthermore, writing out the memberwise copy by hand is tedious and error-prone for classes with many data members (§10.4.6.3).

I use a reference argument for the copy constructor because I must. The copy constructor defines what copying means – including what copying an argument means – so writing

```
complex::complex(complex c) : re(c.re), im(c.im) { }  // error
```

is an error because any call would have involved an infinite recursion.

For other functions taking *complex* arguments, I use value arguments rather than reference arguments. Here, the designer has a choice. From a user's point of view, there is little difference between a function that takes a *complex* argument and one that takes a *const complex&* argument. This issue is discussed further in §11.6.

In principle, copy constructors are used in simple initializations such as

```
complex x = 2;                // create complex(2); then initialize x with it
complex y = complex(2,0);     // create complex(2,0); then initialize y with it
```

However, the calls to the copy constructor are trivially optimized away. We could equivalently have written:

```
complex x(2);       // initialize x by 2
complex y(2,0);     // initialize x by (2,0)
```

For arithmetic types, such as *complex*, I like the look of the version using = better. It is possible to restrict the set of values accepted by the = style of initialization compared to the ()style by making the copy constructor private (§11.2.2) or by declaring a constructor *explicit* (§11.7.1).

Similar to initialization, assignment of two objects of the same class is by default defined as memberwise assignment (§10.2.5). We could explicitly define *complex::operator=* to do that. However, for a simple type like *complex* there is no reason to do so. The default is just right.

The copy constructor – whether user-defined or compiler-generated – is used not only for the initialization of variables, but also for argument passing, value return, and exception handling (see §11.7). The semantics of these operations is defined to be the semantics of initialization (§7.1, §7.3, §14.2.1).

11.3.5 Constructors and Conversions [over.conv]

We defined three versions of each of the four standard arithmetic operators:

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);
// ...
```

This can get tedious, and what is tedious easily becomes error-prone. What if we had three alternatives for the type of each argument for each function? We would need three versions of each single-argument function, nine versions of each two-argument function, twenty-seven versions of each three-argument function, etc. Often these variants are very similar. In fact, almost all variants involve a simple conversion of arguments to a common type followed by a standard algorithm.

The alternative to providing different versions of a function for each combination of arguments is to rely on conversions. For example, our *complex* class provides a constructor that converts a *double* to a *complex*. Consequently, we could simply declare only one version of the equality operator for *complex*:

```
bool operator==(complex,complex);

void f(complex x, complex y)
{
    x==y;      // means operator==(x,y)
    x==3;      // means operator==(x,complex(3))
    3==y;      // means operator==(complex(3),y)
}
```

There can be reasons for preferring to define separate functions. For example, in some cases the conversion can impose overheads, and in other cases, a simpler algorithm can be used for specific argument types. Where such issues are not significant, relying on conversions and providing only the most general variant of a function – plus possibly a few critical variants – contains the combinatorial explosion of variants that can arise from mixed-mode arithmetic.

Where several variants of a function or an operator exist, the compiler must pick ‘‘the right’’ variant based on the argument types and the available (standard and user-defined) conversions. Unless a best match exists, an expression is ambiguous and is an error (see §7.4).

An object constructed by explicit or implicit use of a constructor is automatic and will be destroyed at the first opportunity (see §10.4.10).

No implicit user-defined conversions are applied to the left-hand side of a `.` (or a `->`). This is the case even when the `.` is implicit. For example:

```
void g(complex z)
{
    3+z;                // ok: complex(3)+z
    3.operator+=(z);    // error: 3 is not a class object
    3+=z;               // error: 3 is not a class object
}
```

Thus, you can express the notion that an operator requires an lvalue as their left-hand operand by making that operator a member.

11.3.6 Literals [over.literals]

It is not possible to define literals of a class type in the sense that *1.2* and *12e3* are literals of type *double*. However, literals of the basic types can often be used instead if class member functions are used to provide an interpretation for them. Constructors taking a single argument provide a general mechanism for this. When constructors are simple and inline, it is quite reasonable to think of constructor invocations with literal arguments as literals. For example, I think of *complex(3)* as a literal of type *complex*, even though technically it isn't.

11.3.7 Additional Member Functions [over.additional]

So far, we have provided class *complex* with constructors and arithmetic operators only. That is not quite sufficient for real use. In particular, we often need to be able to examine the value of the real and imaginary parts:

```
class complex {
    double re, im;
public:
    double real() const { return re; }
    double imag() const { return im; }
    // ...
};
```

Unlike the other members of *complex*, *real()* and *imag()* do not modify the value of a *complex*, so they can be declared *const*.

Given *real()* and *imag()*, we can define all kinds of useful operations without granting them direct access to the representation of *complex*. For example:

```
inline bool operator==(complex a, complex b)
{
    return a.real()==b.real() && a.imag()==b.imag();
}
```

Note that we need only to be able to read the real and imaginary parts; writing them is less often needed. If we must do a ‘‘partial update,’’ we can:

```
void f(complex& z, double d)
{
    // ...
    z = complex(z.real(),d);  // assign d to z.im
}
```

A good optimizer generates a single assignment for that statement.

11.3.8 Helper Functions [over.helpers]

If we put all the bits and pieces together, the *complex* class becomes:

```
class complex {
      double re, im;
public:
      complex(double r =0, double i =0) : re(r), im(i) { }

      double real() const { return re; }
      double imag() const { return im; }

      complex& operator+=(complex);
      complex& operator+=(double);
      // –=, *=, and /=
};
```

In addition, we must provide a number of helper functions:

```
complex operator+(complex,complex);
complex operator+(complex,double);
complex operator+(double,complex);

// –, *, and /

complex operator-(complex);   // unary minus
complex operator+(complex);   // unary plus

bool operator==(complex,complex);
bool operator!=(complex,complex);

istream& operator>>(istream&,complex&); // input
ostream& operator<<(ostream&,complex); // output
```

Note that the members *real*() and *imag*() are essential for defining the comparisons. The definition of most of the following helper functions similarly relies on *real*() and *imag*().

We might provide functions to allow users to think in terms of polar coordinates:

```
complex polar(double rho, double theta);
complex conj(complex);

double abs(complex);
double arg(complex);
double norm(complex);

double real(complex);     // for notational convenience
double imag(complex);     // for notational convenience
```

Finally, we must provide an appropriate set of standard mathematical functions:

```
complex acos(complex);
complex asin(complex);
complex atan(complex);
// ...
```

From a user's point of view, the complex type presented here is almost identical to the *complex<double>* found in *<complex>* in the standard library (§22.5).

11.4 Conversion Operators [over.conversion]

Using a constructor to specify type conversion is convenient but has implications that can be undesirable. A constructor cannot specify

[1] an implicit conversion from a user-defined type to a basic type (because the basic types are not classes), or

[2] a conversion from a new class to a previously defined class (without modifying the declaration for the old class).

These problems can be handled by defining a *conversion operator* for the source type. A member function *X::operator T()*, where *T* is a type name, defines a conversion from *X* to *T*. For example, one could define a 6-bit non-negative integer, *Tiny*, that can mix freely with integers in arithmetic operations:

```
class Tiny {
        char v;
        void assign(int i) { if (i&~077) throw Bad_range(); v=i; }
public:
        class Bad_range { };

        Tiny(int i) { assign(i); }
        Tiny& operator=(int i) { assign(i); return *this; }

        operator int() const { return v; }      // conversion to int function
};
```

The range is checked whenever a *Tiny* is initialized by an *int* and whenever an *int* is assigned to one. No range check is needed when we copy a *Tiny*, so the default copy constructor and assignment are just right.

To enable the usual integer operations on *Tiny* variables, we define the implicit conversion from *Tiny* to *int*, *Tiny::operator int()*. Note that the type being converted to is part of the name of the operator and cannot be repeated as the return value of the conversion function:

```
Tiny::operator int() const { return v; }          // right
int Tiny::operator int() const { return v; }      // error
```

In this respect also, a conversion operator resembles a constructor.

Whenever a *Tiny* appears where an *int* is needed, the appropriate *int* is used. For example:

```
int main()
{
        Tiny c1 = 2;
        Tiny c2 = 62;
        Tiny c3 = c2-c1;        // c3 = 60
        Tiny c4 = c3;           // no range check (not necessary)
        int i = c1+c2;          // i = 64

        c1 = c1+c2;             // range error: c1 can't be 64
        i = c3-64;              // i = −4
        c2 = c3-64;             // range error: c2 can't be −4
        c3 = c4;                // no range check (not necessary)
}
```

Conversion functions appear to be particularly useful for handling data structures when reading (implemented by a conversion operator) is trivial, while assignment and initialization are distinctly less trivial.

The ***istream*** and ***ostream*** types rely on a conversion function to enable statements such as

```
while (cin>>x) cout<<x;
```

The input operation ***cin>>x*** returns an ***istream&***. That value is implicitly converted to a value indicating the state of ***cin***. This value can then be tested by the ***while*** (see §21.3.3). However, it is typically *not* a good idea to define an implicit conversion from one type to another in such a way that information is lost in the conversion.

In general, it is wise to be sparing in the introduction of conversion operators. When used in excess, they lead to ambiguities. Such ambiguities are caught by the compiler, but they can be a nuisance to resolve. Probably the best idea is initially to do conversions by named functions, such as ***X::make_int()***. If such a function becomes popular enough to make explicit use inelegant, it can be replaced by a conversion operator ***X::operator int()***.

If both user-defined conversions and user-defined operators are defined, it is possible to get ambiguities between the user-defined operators and the built-in operators. For example:

```
int operator+(Tiny,Tiny);

void f(Tiny t, int i)
{
        t+i;   // error, ambiguous: operator+(t,Tiny(i)) or int(t)+i ?
}
```

It is therefore often best to rely on user-defined conversions or user-defined operators for a given type, but not both.

11.4.1 Ambiguities [over.ambig]

An assignment of a value of type *V* to an object of class *X* is legal if there is an assignment operator *X::operator=(Z)* so that *V* is *Z* or there is a unique conversion of *V* to *Z*. Initialization is treated equivalently.

In some cases, a value of the desired type can be constructed by repeated use of constructors or conversion operators. This must be handled by explicit conversions; only one level of user-defined implicit conversion is legal. In some cases, a value of the desired type can be constructed in more than one way; such cases are illegal. For example:

```
class X { /* ... */ X(int); X(char*); };
class Y { /* ... */ Y(int); };
class Z { /* ... */ Z(X); };

X f(X);
Y f(Y);

Z g(Z);
```

```
void k1()
{
      f(1);            // error: ambiguous f(X(1)) or f(Y(1))?
      f(X(1));         // ok
      f(Y(1));         // ok

      g("Mack");       // error: two user-defined conversions needed; g(Z(X("Mack"))) not tried
      g(X("Doc"));     // ok: g(Z(X("Doc")))
      g(Z("Suzy"));    // ok: g(Z(X("Suzy")))
}
```

User-defined conversions are considered only if they are necessary to resolve a call. For example:

```
class XX { /* ... */ XX(int); };

void h(double);
void h(XX);

void k2()
{
      h(1);        // h(double(1)) or h(XX(1))? h(double(1))!
}
```

The call *h(1)* means *h(double(1))* because that alternative uses only a standard conversion rather than a user-defined conversion (§7.4).

The rules for conversion are neither the simplest to implement, the simplest to document, nor the most general that could be devised. They are, however, considerably safer, and the resulting resolutions are less surprising. It is far easier to manually resolve an ambiguity than to find an error caused by an unsuspected conversion.

The insistence on strict bottom-up analysis implies that the return type is not used in overloading resolution. For example:

```
class Quad {
public:
      Quad(double);
      // ...
};

Quad operator+(Quad,Quad);

void f(double a1, double a2)
{
      Quad r1 = a1+a2;              // double-precision add
      Quad r2 = Quad(a1)+a2;        // force quad arithmetic
}
```

The reason for this design choice is partly that strict bottom-up analysis is more comprehensible and partly that it is not considered the compiler's job to decide which precision the programmer might want for the addition.

Once the types of both sides of an initialization or assignment have been determined, both types are used to resolve the initialization or assignment. For example:

```
class Real {
public:
      operator double();
      operator int();
      // ...
};

void g(Real a)
{
      double d = a;    // d = a.double();
      int i = a;       // i = a.int();

      d = a;           // d = a.double();
      i = a;           // i = a.int();
}
```

In these cases, the type analysis is still bottom-up, with only a single operator and its argument types considered at any one time.

11.5 Friends [over.friends]

An ordinary member function declaration specifies three logically distinct things:

[1] The function can access the private part of the class declaration, and
[2] the function is in the scope of the class, and
[3] the function must be invoked on an object (has a *this* pointer).

By declaring a member function *static* (§10.2.4), we can give it the first two properties only. By declaring a function a *friend*, we can give it the first property only.

For example, we could define an operator that multiplies a *Matrix* by a *Vector*. Naturally, *Vector* and *Matrix* each hide their representation and provide a complete set of operations for manipulating objects of their type. However, our multiplication routine cannot be a member of both. Also, we don't really want to provide low-level access functions to allow every user to both read and write the complete representation of both *Matrix* and *Vector*. To avoid this, we declare the *operator** a friend of both:

```
class Matrix;

class Vector {
      float v[4];
      // ...
      friend Vector operator*(const Matrix&, const Vector&);
};

class Matrix {
      Vector v[4];
      // ...
      friend Vector operator*(const Matrix&, const Vector&);
};
```

```
Vector operator*(const Matrix& m, const Vector& v)
{
        Vector r;
        for (int i = 0; i<4; i++) {        // r[i] = m[i] * v;
                r.v[i] = 0;
                for (int j = 0; j<4; j++) r.v[i] += m.v[i].v[j] * v.v[j];
        }
        return r;
}
```

A *friend* declaration can be placed in either the private or the public part of a class declaration; it does not matter where. Like a member function, a friend function is explicitly declared in the declaration of the class of which it is a friend. It is therefore as much a part of that interface as is a member function.

A member function of one class can be the friend of another. For example:

```
class List_iterator {
        // ...
        int* next();
};

class List {
        friend int* List_iterator::next();
        // ...
};
```

It is not unusual for all functions of one class to be friends of another. There is a shorthand for this:

```
class List {
        friend class List_iterator;
        // ...
};
```

This friend declaration makes all of ***List_iterator***'s member functions friends of ***List***.

Clearly, *friend* classes should be used only to express closely connected concepts. Often, there is a choice between making a class a member (a nested class) or a friend (§24.4).

11.5.1 Finding Friends [over.lookup]

Like a member declaration, a *friend* declaration does not introduce a name into an enclosing scope. For example:

```
class Matrix {
        friend class Xform;
        friend Matrix invert(const Matrix&);
        // ...
};

Xform x;                                // error: no Xform in scope
Matrix (*p)(const Matrix&) = &invert;   // error: no invert() in scope
```

For large programs and large classes, it is nice that a class doesn't ‘‘quietly’’ add names to its enclosing scope. For a template class that can be instantiated in many different contexts (Chapter 13), this is very important.

A friend class must be previously declared in an enclosing scope or defined in the non-class scope immediately enclosing the class that is declaring it a friend. For example:

```
class X { /* ... */ };              // Y's friend

namespace N {
      class Y {
            friend class X;
            friend class Z;
            friend class AE;
      };
      class Z { /* ... */ };        // Y's friend
}

class AE { /* ... */ };             // not a friend of Y
```

A friend function can be explicitly declared just like friend classes, or it can be found through its argument types (§8.2.6) as if it was declared in the non-class scope immediately enclosing its class. For example:

```
void f(Matrix& m)
{
      invert(m);        // Matrix's friend invert()
}
```

It follows that a friend function should either be explicitly declared in an enclosing scope or take an argument of its class. If not, the friend cannot be called. For example:

```
// no f() here

void g();                           // X's friend

class X {
      friend void f();              // useless
      friend void g();
      friend void h(const X&);      // can be found through its argument
};

void f() { /* ... */ }              // not a friend of X
```

11.5.2 Friends and Members [over.friends.members]

When should we use a friend function, and when is a member function the better choice for specifying an operation? First, we try to minimize the number of functions that access the representation of a class and try to make the set of access functions as appropriate as possible. Therefore, the first question is not, ‘‘Should it be a member, a static member, or a friend?’’ but rather, ‘‘Does it really need access?’’ Typically, the set of functions that need access is smaller than we are willing to believe at first.

Some operations must be members – for example, constructors, destructors, and virtual functions (§12.2.6) – but typically there is a choice. Because member names are local to the class, a function should be a member unless there is a specific reason for it to be a nonmember.

Consider a class *X* presenting alternative ways of presenting an operation:

```
class X {
    // ...
    X(int);

    int m1();
    int m2() const;

    friend int f1(X&);
    friend int f2(const X&);
    friend int f3(X);
};
```

Member functions can be invoked for objects of their class only; no user-defined conversions are applied. For example:

```
void g()
{
    99.m1(); // error: X(99).m1() not tried
    99.m2(); // error: X(99).m2() not tried
}
```

The conversion *X(int)* is not applied to make an *X* out of *99*.

The global function *f1()* has a similar property because implicit conversions are not used for non-*const* reference arguments (§5.5, §11.3.5). However, conversions may be applied to the arguments of *f2()* and *f3()*:

```
void h()
{
    f1(99);    // error: f1(X(99)) not tried
    f2(99);    // ok: f2(X(99));
    f3(99);    // ok: f3(X(99));
}
```

An operation modifying the state of a class object should therefore be a member or a global function taking a non-*const* reference argument (or a non-*const* pointer argument). Operators that require lvalue operands for the fundamental types (=, *=, ++, etc.) are most naturally defined as members for user-defined types.

Conversely, if implicit type conversion is desired for all operands of an operation, the function implementing it must be a nonmember function taking a *const* reference argument or a non-reference argument. This is often the case for the functions implementing operators that do not require lvalue operands when applied to fundamental types (+, -, ||, etc.). Such operators often need access to the representations of their operand class. Consequently, binary operators are the most common source of *friend* functions.

If no type conversions are defined, there appears to be no compelling reason to choose a member over a friend taking a reference argument, or vice versa. In some cases, the programmer may have a preference for one call syntax over another. For example, most people seem to prefer the notation *inv(m)* for inverting a ***Matrix m*** to the alternative ***m.inv()***. Naturally, if ***inv()*** really does invert ***m*** itself, rather than return a new ***Matrix*** that is the inverse of ***m***, it should be a member.

All other things considered equal, choose a member. It is not possible to know if someone someday will define a conversion operator. It is not always possible to predict if a future change may require changes to the state of the object involved. The member function call syntax makes it clear to the user that the object may be modified; a reference argument is far less obvious. Furthermore, expressions in the body of a member can be noticeably shorter than the equivalent expressions in a global function; a nonmember function must use an explicit argument, whereas the member can use ***this*** implicitly. Also, because member names are local to the class they tend to be shorter than the names of nonmember functions.

11.6 Large Objects [over.large]

We defined the *complex* operators to take arguments of type *complex*. This implies that for each use of a *complex* operator, each operand is copied. The overhead of copying two ***doubles*** can be noticeable but often less than what a pair of pointers impose. Unfortunately, not all classes have a conveniently small representation. To avoid excessive copying, one can declare functions to take reference arguments. For example:

```
class Matrix {
      double m[4][4];
public:
      Matrix();
      friend Matrix operator+(const Matrix&, const Matrix&);
      friend Matrix operator*(const Matrix&, const Matrix&);
};
```

References allow the use of expressions involving the usual arithmetic operators for large objects without excessive copying. Pointers cannot be used because it is not possible to redefine the meaning of an operator applied to a pointer. Addition could be defined like this:

```
Matrix operator+(const Matrix& arg1, const Matrix& arg2)
{
      Matrix sum;
      for (int i=0; i<4; i++)
            for (int j=0; j<4; j++)
                  sum.m[i][j] = arg1.m[i][j] + arg2.m[i][j];
      return sum;
}
```

This *operator+()* accesses the operands of + through references but returns an object value. Returning a reference would appear to be more efficient:

```
class Matrix {
      // ...
      friend Matrix& operator+(const Matrix&, const Matrix&);
      friend Matrix& operator*(const Matrix&, const Matrix&);
};
```

This is legal, but it causes a memory allocation problem. Because a reference to the result will be passed out of the function as a reference to the return value, the return value cannot be an automatic variable (§7.3). Since an operator is often used more than once in an expression, the result cannot be a *static* local variable. The result would typically be allocated on the free store. Copying the return value is often cheaper (in execution time, code space, and data space) than allocating and (eventually) deallocating an object on the free store. It is also much simpler to program.

There are techniques you can use to avoid copying the result. The simplest is to use a buffer of static objects. For example:

```
const max_matrix_temp = 7;

Matrix& get_matrix_temp()
{
        static int nbuf = 0;
        static Matrix buf[max_matrix_temp];

        if (nbuf == max_matrix_temp) nbuf = 0;
        return buf[nbuf++];
}

Matrix& operator+(const Matrix& arg1, const Matrix& arg2)
{
        Matrix& res = get_matrix_temp();
        // ...
        return res;
}
```

Now a *Matrix* is copied only when the result of an expression is assigned. However, heaven help you if you write an expression that involves more than *max_matrix_temp* temporaries!

A less error-prone technique involves defining the matrix type as a handle (§25.7) to a representation type that really holds the data. In that way, the matrix handles can manage the representation objects in such a way that allocation and copying are minimized (see §11.12 and §11.14[18]). However, that strategy relies on operators returning objects rather than references or pointers. Another technique is to define ternary operations and have them automatically invoked for expressions such as *a=b+c* and *a+b*i* (§21.4.6.3, §22.4.7).

11.7 Essential Operators [over.essential]

In general, for a type *X*, the copy constructor *X(const X&)* takes care of initialization by an object of the same type *X*. It cannot be overemphasized that *assignment and initialization are different operations* (§10.4.4.1). This is especially important when a destructor is declared. If a class *X* has a destructor that performs a nontrivial task, such as free-store deallocation, the class is likely to need the full complement of functions that control construction, destruction, and copying:

```
class X {
        // ...
        X(Sometype);                // constructor: create objects
        X(const X&);                // copy constructor
        X& operator=(const X&);     // copy assignment: cleanup and copy
        ~X();                       // destructor: cleanup
};
```

There are three more cases in which an object is copied: as a function argument, as a function return value, and as an exception. When an argument is passed, a hitherto uninitialized variable – the formal parameter – is initialized. The semantics are identical to those of other initializations. The same is the case for function return values and exceptions, although that is less obvious. In such cases, the copy constructor will be applied. For example:

```
string g(string arg)
{
        return arg;
}

int main ()
{
        string s = "Newton";
        s = g(s);
}
```

Clearly, the value of *s* ought to be *"Newton"* after the call of *g()*. Getting a copy of the value of *s* into the argument *arg* is not difficult; a call of *string*'s copy constructor does that. Getting a copy of that value out of *g()* takes another call of *string(const string&)*; this time, the variable initialized is a temporary one, which is then assigned to *s*. Often one, but not both, of these copy operations can be optimized away. Such temporary variables are, of course, destroyed properly using *string::~string()* (see §10.4.10).

For a class *X* for which the assignment operator *X::operator=(const X&)* and the copy constructor *X::X(const X&)* are not explicitly declared by the programmer, the missing operation or operations will be generated by the compiler (§10.2.5).

11.7.1 Explicit Constructors [over.explicit]

By default, a single argument constructor also defines an implicit conversion. For some types, that is ideal. For example:

```
complex z = 2; // initialize z with complex(2)
```

In other cases, the implicit conversion is undesirable and error-prone. For example:

```
string s = ´a´; // make s a string with int(’a’) elements
```

It is quite unlikely that this was what the person defining *s* meant.

Implicit conversion can be suppressed by declaring a constructor *explicit*. That is, an *explicit* constructor will be invoked only explicitly. In particular, where a copy constructor is in principle needed (§11.3.4), an *explicit* constructor will not be implicitly invoked. For example:

```
class String {
        // ...
        explicit String(int n);        // preallocate n bytes
        String(const char* p);         // initial value is the C-style string p
};

String s1 = ´a´;                       // error: no implicit char–>String conversion
String s2(10);                         // ok: String with space for 10 characters
String s3 = String(10);                // ok: String with space for 10 characters
String s4 = "Brian";                   // ok: s4 = String("Brian")
String s5("Fawlty");

void f(String);

String g()
{
        f(10);                 // error: no implicit int–>String conversion
        f(String(10));
        f("Arthur");           // ok: f(String("Arthur"))
        f(s1);

        String* p1 = new String("Eric");
        String* p2 = new String(10);

        return 10;             // error: no implicit int–>String conversion
}
```

The distinction between

```
String s1 = ´a´;               // error: no implicit char–>String conversion
```

and

```
String s2(10);                 // ok: string with space for 10 characters
```

may seem subtle, but it is less so in real code than in contrived examples.

In *Date*, we used a plain *int* to represent a year (§10.3). Had *Date* been critical in our design, we might have introduced a *Year* type to allow stronger compile-time checking. For example:

```
class Year {
        int y;
public:
        explicit Year(int i) : y(i) { }          // construct Year from int
        operator int() const { return y; }       // conversion: Year to int
};

class Date {
public:
        Date(int d, Month m, Year y);
        // ...
};

Date d3(1978,feb,21);          // error: 21 is not a Year
Date d4(21,feb,Year(1978));    // ok
```

The *Year* class is a simple ‘‘wrapper’’ around an *int*. Thanks to the *operator int*( ), a *Year* is implicitly converted into an *int* wherever needed. By declaring the constructor *explicit*, we make sure that the *int* to *Year* happens only when we ask for it and that ‘‘accidental’’ assignments are caught at compile time. Because *Year*’s member functions are easily inlined, no run-time or space costs are added.

A similar technique can be used to define range types (§25.6.1).

11.8 Subscripting [over.subscript]

An *operator*[ ] function can be used to give subscripts a meaning for class objects. The second argument (the subscript) of an *operator*[ ] function may be of any type. This makes it possible to define *vector*s, associative arrays, etc.

As an example, let us recode the example from §5.5 in which an associative array is used to write a small program for counting the number of occurrences of words in a file. There, a function is used. Here, an associative array type is defined:

```
class Assoc {
        struct Pair {
                string name;
                double val;
                Pair(string n ="", double v =0) :name(n), val(v) { }
        };
        vector<Pair> vec;

        Assoc(const Assoc&);                    // private to prevent copying
        Assoc& operator=(const Assoc&);         // private to prevent copying
public:
        Assoc() {}
        double& operator[](const string&);
        void print_all() const;
};
```

An *Assoc* keeps a vector of *Pair*s. The implementation uses the same trivial and inefficient search method as in §5.5:

```
double& Assoc::operator[](const string& s)
        // search for s; return its value if found; otherwise, make a new Pair and return the default value 0
{
        for (vector<Pair>::const_iterator p = vec.begin(); p!=vec.end(); ++p)
                if (s == p->name) return p->val;

        vec.push_back(Pair(s,0));       // initial value: 0

        return vec.back().val;          // return last element (§16.3.3)
}
```

Because the representation of an *Assoc* is hidden, we need a way of printing it:

```
void Assoc::print_all() const
{
    for (vector<Pair>::const_iterator p = vec.begin(); p!=vec.end(); ++p)
        cout << p->name << ": " << p->val << ´\n´;
}
```

Finally, we can write the trivial main program:

```
int main()      // count the occurrences of each word on input
{
    string buf;
    Assoc vec;
    while (cin>>buf) vec[buf]++;
    vec.print_all();
}
```

A further development of the idea of an associative array can be found in §17.4.1.

An *operator*[]() must be a member function.

11.9 Function Call [over.call]

Function call, that is, the notation *expression(expression-list)*, can be interpreted as a binary operation with the *expression* as the left-hand operand and the *expression-list* as the right-hand operand. The call operator () can be overloaded in the same way as other operators can. An argument list for an *operator*()() is evaluated and checked according to the usual argument-passing rules. Overloading function call seems to be useful primarily for defining types that have only a single operation and for types for which one operation is predominant.

The most obvious, and probably also the most important, use of the () operator is to provide the usual function call syntax for objects that in some way behave like functions. An object that acts like a function is often called a *function-like object* or simply a *function object* (§18.4). Such function objects are important because they allow us to write code that takes nontrivial operations as parameters. For example, the standard library provides many algorithms that invoke a function for each element of a container. Consider:

```
void negate(complex& c) { c = -c; }

void f(vector<complex>& aa, list<complex>& ll)
{
    for_each(aa.begin(),aa.end(),negate);  // negate all vector elements
    for_each(ll.begin(),ll.end(),negate);  // negate all list elements
}
```

This negates every element in the vector and the list.

What if we wanted to add *complex*(2,3) to every element? That is easily done like this:

```
void add23(complex& c)
{
        c += complex(2,3);
}

void g(vector<complex>& aa, list<complex>& ll)
{
        for_each(aa.begin(),aa.end(),add23);
        for_each(ll.begin(),ll.end(),add23);
}
```

How would we write a function to repeatedly add an arbitrary complex value? We need something to which we can pass that arbitrary value and which can then use that value each time it is called. That does not come naturally for functions. Typically, we end up ‘‘passing’’ the arbitrary value by leaving it in the function’s surrounding context. That’s messy. However, we can write a class that behaves in the desired way:

```
class Add {
        complex val;
public:
        Add(complex c) { val = c; }                              // save value
        Add(double r, double i) { val = complex(r,i); }

        void operator()(complex& c) const { c += val; }          // add value to argument
};
```

An object of class `Add` is initialized with a complex number, and when invoked using `()`, it adds that number to its argument. For example:

```
void h(vector<complex>& aa, list<complex>& ll, complex z)
{
        for_each(aa.begin(),aa.end(),Add(2,3));
        for_each(ll.begin(),ll.end(),Add(z));
}
```

This will add `complex(2,3)` to every element of the array and `z` to every element on the list. Note that `Add(z)` constructs an object that is used repeatedly by `for_each()`. It is not simply a function that is called once or even called repeatedly. The function that is called repeatedly is `Add(z)`’s `operator()()`.

This all works because `for_each` is a template that applies `()` to its third argument without caring exactly what that third argument really is:

```
template<class Iter, class Fct> Iter for_each(Iter b, Iter e, Fct f)
{
        while (b != e) f(*b++);
        return b;
}
```

At first glance, this technique may look esoteric, but it is simple, efficient, and extremely useful (see §3.8.5, §18.4).

Other popular uses of *operator*()() are as a substring operator and as a subscripting operator for multidimensional arrays (§22.4.5).

An *operator*()() must be a member function.

11.10 Dereferencing [over.deref]

The dereferencing operator -> can be defined as a unary postfix operator. That is, given a class

```
class Ptr {
        // ...
        X* operator->();
};
```

objects of class *Ptr* can be used to access members of class *X* in a very similar manner to the way pointers are used. For example:

```
void f(Ptr p)
{
        p->m = 7;       // (p.operator–>())–>m = 7
}
```

The transformation of the object *p* into the pointer *p.operator*->() does not depend on the member *m* pointed to. That is the sense in which *operator*->() is a unary postfix operator. However, there is no new syntax introduced, so a member name is still required after the ->. For example:

```
void g(Ptr p)
{
        X* q1 = p->;                  // syntax error
        X* q2 = p.operator->();       // ok
}
```

Overloading -> is primarily useful for creating ‘‘smart pointers,’’ that is, objects that act like pointers and in addition perform some action whenever an object is accessed through them. For example, one could define a class *Rec_ptr* for accessing objects of class *Rec* stored on disk. *Rec_ptr*’s constructor takes a name that can be used to find the object on disk, *Rec_ptr*::*operator*->() brings the object into main memory when accessed through its *Rec_ptr*, and *Rec_ptr*’s destructor eventually writes the updated object back out to disk:

```
class Rec_ptr {
        Rec* in_core_address;
        const char* identifier;
        // ...

public:
        Rec_ptr(const char* p) : identifier(p), in_core_address(0) { }
        ~Rec_ptr() { write_to_disk(in_core_address,identifier); }
        Rec* operator->();
};
```

```
Rec* Rec_ptr::operator->()
{
      if (in_core_address == 0) in_core_address = read_from_disk(identifier);
      return in_core_address;
}
```

Rec_ptr might be used like this:

```
struct Rec {     // the Rec that a Rec_ptr points to
      string name;
      // ...
};

void update(const char* s)
{
      Rec_ptr p(s);                  // get Rec_ptr for s

      p->name = "Roscoe";            // update s; if necessary, first retrieve from disk
      // ...
}
```

Naturally, a real *Rec_ptr* would be a template so that the *Rec* type is a parameter. Also, a realistic program would contain error-handling code and use a less naive way of interacting with the disk.

For ordinary pointers, use of `->` is synonymous with some uses of unary `*` and `[]`. Given

```
Y* p;
```

it holds that

```
p->m == (*p).m == p[0].m
```

As usual, no such guarantee is provided for user-defined operators. The equivalence can be provided where desired:

```
class Ptr_to_Y {
      Y* p;
public:
      Y* operator->() { return p; }
      Y& operator*() { return *p; }
      Y& operator[](int i) { return p[i]; }
};
```

If you provide more than one of these operators, it might be wise to provide the equivalence, just as it is wise to ensure that *++x* and *x+=1* have the same effect as *x=x+1* for a simple variable *x* of some class if ++, +=, =, and + are provided.

The overloading of `->` is important to a class of interesting programs and not just a minor curiosity. The reason is that *indirection* is a key concept and that overloading `->` provides a clean, direct, and efficient way of representing indirection in a program. Iterators (Chapter 19) provide an important example of this. Another way of looking at operator `->` is to consider it as a way of providing C++ with a limited, but useful, form of *delegation* (§24.2.4).

Operator `->` must be a member function. If used, its return type must be a pointer or an object of a class to which you can apply `->`. When declared for a template class, *operator*`->()` is frequently unused, so it makes sense to postpone checking the constraint on the return type until actual use.

11.11 Increment and Decrement [over.incr]

Once people invent ‘‘smart pointers,’’ they often decide to provide the increment operator ++ and the decrement operator -- to mirror these operators’ use for built-in types. This is especially obvious and necessary where the aim is to replace an ordinary pointer type with a ‘‘smart pointer’’ type that has the same semantics, except that it adds a bit of run-time error checking. For example, consider a troublesome traditional program:

```
void f1(T a)          // traditional use
{
    T v[200];
    T* p = &v[0];
    p--;
    *p = a;     // Oops: ‘p’ out of range, uncaught
    ++p;
    *p = a;     // ok
}
```

We might want to replace the pointer *p* with an object of a class *Ptr_to_T* that can be dereferenced only provided it actually points to an object. We would also like to ensure that *p* can be incremented and decremented, only provided it points to an object within an array and the increment and decrement operations yield an object within the array. That is we would like something like this:

```
class Ptr_to_T {
    // ...
};

void f2(T a)          // checked
{
    T v[200];
    Ptr_to_T p(&v[0],v,200);
    p--;
    *p = a;     // run-time error: ‘p’ out of range
    ++p;
    *p = a;     // ok
}
```

The increment and decrement operators are unique among C++ operators in that they can be used as both prefix and postfix operators. Consequently, we must define prefix and postfix increment and decrement *Ptr_to_T*. For example:

```
class Ptr_to_T {
        T* p;
        T* array;
        int size;
public:

        Ptr_to_T(T* p, T* v, int s);     // bind to array v of size s, initial value p
        Ptr_to_T(T* p);                  // bind to single object, initial value p

        Ptr_to_T& operator++();          // prefix
        Ptr_to_T operator++(int);        // postfix

        Ptr_to_T& operator--();          // prefix
        Ptr_to_T operator--(int);        // postfix

        T& operator*();         // prefix
};
```

The *int* argument is used to indicate that the function is to be invoked for postfix application of ++. This *int* is never used; the argument is simply a dummy used to distinguish between prefix and postfix application. The way to remember which version of an *operator++* is prefix is to note that the version without the dummy argument is prefix, exactly like all the other unary arithmetic and logical operators. The dummy argument is used only for the ‘‘odd’’ postfix ++ and --.

Using *Ptr_to_T*, the example is equivalent to:

```
void f3(T a)            // checked
{
        T v[200];
        Ptr_to_T p(&v[0],v,200);
        p.operator--(0);
        p.operator*() = a;  // run-time error: 'p' out of range
        p.operator++();
        p.operator*() = a;  // ok
}
```

Completing class *Ptr_to_T* is left as an exercise (§11.14[19]). Its elaboration into a template using exceptions to report the run-time errors is another exercise (§14.12[2]). An example of operators ++ and -- for iteration can be found in §19.3. A pointer template that behaves correctly with respect to inheritance is presented in (§13.6.3).

11.12 A String Class [over.string]

Here is a more realistic version of class *String*. I designed it as the minimal string that served my needs. This string provides value semantics, character read and write operations, checked and unchecked access, stream I/O, literal strings as literals, and equality and concatenation operators. It represents strings as C-style, zero-terminated arrays of characters and uses reference counts to minimize copying. Writing a better string class and/or one that provides more facilities is a good exercise (§11.14[7-12]). That done, we can throw away our exercises and use the standard library string (Chapter 20).

My almost-real *String* employs three auxiliary classes: *Srep*, to allow an actual representation to be shared between several *String*s with the same value; *Range*, to be thrown in case of range errors, and *Cref*, to help implement a subscript operator that distinguishes between reading and writing:

```
class String {
        struct Srep;                    // representation
        Srep *rep;
public:
        class Cref;                     // reference to char

        class Range { };                // for exceptions

        // ...
};
```

Like other members, a *member class* (often called a *nested class*) can be declared in the class itself and defined later:

```
struct String::Srep {
        char* s;                // pointer to elements
        int sz;                 // number of characters
        int n;                  // reference count

        Srep(int nsz, const char* p)
        {
                n = 1;
                sz = nsz;
                s = new char[sz+1]; // add space for terminator
                strcpy(s,p);
        }

        ~Srep() { delete[] s; }

        Srep* get_own_copy()            // clone if necessary
        {
                if (n==1) return this;
                n--;
                return new Srep(sz,s);
        }

        void assign(int nsz, const char* p)
        {
                if (sz != nsz) {
                        delete[] s;
                        sz = nsz;
                        s = new char[sz+1];
                }
                strcpy(s,p);
        }
private:                    // prevent copying:
        Srep(const Srep&);
        Srep& operator=(const Srep&);
};
```

Class *String* provides the usual set of constructors, destructor, and assignment operations:

```
class String {
        // ...

        String();                       // x = ""
        String(const char*);            // x = "abc"
        String(const String&);          // x = other_string
        String& operator=(const char *);
        String& operator=(const String&);
        ~String();

        // ...
};
```

This *String* has value semantics. That is, after an assignment *s1=s2*, the two strings *s1* and *s2* are fully distinct and subsequent changes to the one have no effect on the other. The alternative would be to give *String* pointer semantics. That would be to let changes to *s2* after *s1=s2* also affect the value of *s1*. For types with conventional arithmetic operations, such as complex, vector, matrix, and string, I prefer value semantics. However, for the value semantics to be affordable, a *String* is implemented as a handle to its representation and the representation is copied only when necessary:

```
String::String()                // the empty string is the default value
{
        rep = new Srep(0,"");
}

String::String(const String& x) // copy constructor
{
        x.rep->n++;
        rep = x.rep;    // share representation
}

String::~String()
{
        if (--rep->n == 0) delete rep;
}

String& String::operator=(const String& x)      // copy assignment
{
        x.rep->n++;                     // protects against ‘‘st = st’’
        if (--rep->n == 0) delete rep;
        rep = x.rep;                    // share representation
        return *this;
}
```

Pseudo-copy operations taking *const char** arguments are provided to allow string literals:

```
String::String(const char* s)
{
    rep = new Srep(strlen(s),s);
}

String& String::operator=(const char* s)
{
    if (rep->n == 1)                    // recycle Srep
        rep->assign(strlen(s),s);
    else {                              // use new Srep
        rep->n--;
        rep = new Srep(strlen(s),s);
    }
    return *this;
}
```

The design of access operators for a string is a difficult topic because ideally access is by conventional notation (that is, using `[]`), maximally efficient, and range checked. Unfortunately, you cannot have all of these properties simultaneously. My choice here has been to provide efficient unchecked operations with a slightly inconvenient notation plus slightly less efficient checked operators with the conventional notation:

```
class String {
    // ...

    void check(int i) const { if (i<0 || rep->sz<=i) throw Range(); }

    char read(int i) const { return rep->s[i]; }
    void write(int i, char c) { rep=rep->get_own_copy(); rep->s[i]=c; }

    Cref operator[](int i) { check(i); return Cref(*this,i); }
    char operator[](int i) const { check(i); return rep->s[i]; }

    int size() const { return rep->sz; }

    // ...
};
```

The idea is to use `[]` to get checked access for ordinary use, but to allow the user to optimize by checking the range once for a set of accesses. For example:

```
int hash(const String& s)
{
    int h = s.read(0);
    const int max = s.size();
    for (int i = 1; i<max; i++) h ^= s.read(i)>>1; // unchecked access to s
    return h;
}
```

Defining an operator, such as `[]`, to be used for both reading and writing is difficult where it is not acceptable simply to return a reference and let the user decide what to do with it. Here, that is not a reasonable alternative because I have defined ***String*** so that the representation is shared between ***String***s that have been assigned, passed as value arguments, etc., until someone actually writes to a

String. Then, and only then, is the representation copied. This technique is usually called *copy-on-write*. The actual copy is done by *String::Srep::get_own_copy()*.

To get these access functions inlined, their definitions must be placed so that the definition of *Srep* is in scope. This implies that either *Srep* is defined within *String* or the access functions are defined *inline* outside *String* and after *String::Srep* (§11.14[2]).

To distinguish between a read and a write, *String::operator[]()* returns a *Cref* when called for a non-*const* object. A *Cref* behaves like a *char&*, except that it calls *String::Srep::get_own_copy()* when written to:

```
class String::Cref {          // reference to s[i]
friend class String;
        String& s;
        int i;
        Cref(String& ss, int ii) : s(ss), i(ii) { }
public:
        operator char() { return s.read(i); }              // yield value
        void operator=(char c) { s.write(i,c); }            // change value
};
```

For example:

```
void f(String s, const String& r)
{
        int c1 = s[1];    // c1 = s.operator[](1).operator char()
        s[1] = 'c';       // s.operator[](1).operator=(’c’)

        int c2 = r[1];    // c2 = r.operator[](1)
        r[1] = 'd';       // error: assignment to char, r.operator[](1) = ’d’
}
```

Note that for a non-*const* object *s.operator[](1)* is *Cref(s,1)*.

To complete class *String*, I provide a set of useful functions:

```
class String {
        // ...

        String& operator+=(const String&);
        String& operator+=(const char*);

        friend ostream& operator<<(ostream&, const String&);
        friend istream& operator>>(istream&, String&);

        friend bool operator==(const String& x, const char* s)
                { return strcmp(x.rep->s, s) == 0; }

        friend bool operator==(const String& x, const String& y)
                { return strcmp(x.rep->s, y.rep->s) == 0; }

        friend bool operator!=(const String& x, const char* s)
                { return strcmp(x.rep->s, s) != 0; }
        friend bool operator!=(const String& x, const String& y)
                { return strcmp(x.rep->s, y.rep->s) != 0; }
};

String operator+(const String&, const String&);
String operator+(const String&, const char*);
```

To save space, I have left the I/O and concatenation operations as exercises.

The main program simply exercises the *String* operators a bit:

```
    String f(String a, String b)
    {
        a[2] = 'x';
        char c = b[3];
        cout << "in f: " << a << ' ' << b << ' ' << c << '\n';
        return b;
    }

    int main()
    {
        String x, y;
        cout << "Please enter two strings\n";
        cin >> x >> y;
        cout << "input: " << x << ' ' << y << '\n';
        String z = x;
        y = f(x,y);
        if (x != z) cout << "x corrupted!\n";
        x[0] = '!';
        if (x == z) cout << "write failed!\n";
        cout << "exit: " << x << ' ' << y << ' ' << z << '\n';
    }
```

This *String* lacks many features that you might consider important or even essential. For example, it offers no operation of producing a C-string representation of its value (§11.14[10], Chapter 20).

11.13 Advice [class.advice]

[1] Define operators primarily to mimic conventional usage; §11.1.
[2] For large operands, use *const* reference argument types; §11.6.
[3] For large results, consider optimizing the return; §11.6.
[4] Prefer the default copy operations if appropriate for a class; §11.3.4.
[5] Redefine or prohibit copying if the default is not appropriate for a type; §11.2.2.
[6] Prefer member functions over nonmembers for operations that need access to the representation; §11.5.2.
[7] Prefer nonmember functions over members for operations that do not need access to the representation; §11.5.2.
[8] Use namespaces to associate helper functions with ‘‘their’’ class; §11.2.4.
[9] Use nonmember functions for symmetric operators; §11.3.2.
[10] Use `()` for subscripting multidimensional arrays; §11.9.

[11] Make constructors that take a single ‘‘size argument’’ ***explicit***; §11.7.1.
[12] For non-specialized uses, prefer the standard ***string*** (Chapter 20) to the result of your own exercises; §11.12.
[13] Be cautious about introducing implicit conversions; §11.4.
[14] Use member functions to express operators that require an lvalue as its left-hand operand; §11.3.5.

11.14 Exercises [over.exercises]

1. (∗2) In the following program, which conversions are used in each expression?

```
struct X {
    int i;
    X(int);
    operator+(int);
};

struct Y {
    int i;
    Y(X);
    operator+(X);
    operator int();
};

extern X operator*(X, Y);
extern int f(X);

X x = 1;
Y y = x;
int i = 2;

int main()
{
    i + 10;       y + 10;       y + 10 * y;
    x + y + i;    x * x + i;    f(7);
    f(y);         y + y;        106 + y;
}
```

   Modify the program so that it will run and print the values of each legal expression.
2. (∗2) Complete and test class ***String*** from §11.12.
3. (∗2) Define a class ***INT*** that behaves exactly like an ***int***. Hint: Define ***INT::operator int()***.
4. (∗1) Define a class ***RINT*** that behaves like an ***int*** except that the only operations allowed are + (unary and binary), - (unary and binary), *, /, and %. Hint: Do not define ***RINT::operator int()***.
5. (∗3) Define a class ***LINT*** that behaves like a ***RINT***, except that it has at least 64 bits of precision.
6. (∗4) Define a class implementing arbitrary precision arithmetic. Test it by calculating the factorial of *1000*. Hint: You will need to manage storage in a way similar to what was done for class ***String***.

7. (∗2) Define an external iterator for class *String*:

```
class String_iter {
        // refer to string and string element
public:
        String_iter(String& s);          // iterator for s
        char& next();                    // reference to next element

        // more operations of your choice
};
```

   Compare this in utility, programming style, and efficiency to having an internal iterator for *String* (that is, a notion of a current element for the *String* and operations relating to that element).
8. (∗1.5) Provide a substring operator for a string class by overloading `()`. What other operations would you like to be able to do on a string?
9. (∗3) Design class *String* so that the substring operator can be used on the left-hand side of an assignment. First, write a version in which a string can be assigned to a substring of the same length. Then, write a version in which the lengths may differ.
10. (∗2) Define an operation for *String* that produces a C-string representation of its value. Discuss the pros and cons of having that operation as a conversion operator. Discuss alternatives for allocating the memory for that C-string representation.
11. (∗2.5) Define and implement a simple regular expression pattern match facility for class *String*.
12. (∗1.5) Modify the pattern match facility from §11.14[11] to work on the standard library *string*. Note that you cannot modify the definition of *string*.
13. (∗2) Write a program that has been rendered unreadable through use of operator overloading and macros. An idea: Define `+` to mean `-` and vice versa for *INTs*. Then, use a macro to define *int* to mean *INT*. Redefine popular functions using reference type arguments. Writing a few misleading comments can also create great confusion.
14. (∗3) Swap the result of §11.14[13] with a friend. Without running it, figure out what your friend's program does. When you have completed this exercise, you'll know what to avoid.
15. (∗2) Define a type *Vec4* as a vector of four *float*s. Define *operator*`[]` for *Vec4*. Define operators `+`, `-`, `*`, `/`, `=`, `+=`, `-=`, `*=`, and `/=` for combinations of vectors and floating-point numbers.
16. (∗3) Define a class *Mat4* as a vector of four *Vec4*s. Define *operator*`[]` to return a *Vec4* for *Mat4*. Define the usual matrix operations for this type. Define a function doing Gaussian elimination for a *Mat4*.
17. (∗2) Define a class *Vector* similar to *Vec4* but with the size given as an argument to the constructor *Vector*`::`*Vector*`(`*int*`)`.
18. (∗3) Define a class *Matrix* similar to *Mat4* but with the dimensions given as arguments to the constructor *Matrix*`::`*Matrix*`(`*int*`,` *int*`)`.
19. (∗2) Complete class *Ptr_to_T* from §11.11 and test it. To be complete, *Ptr_to_T* must have at least the operators `*`, `->`, `=`, `++`, and `--` defined. Do not cause a run-time error until a wild pointer is actually dereferenced.

20. (∗1) Given two structures:

```
struct S { int x, y; };
struct T { char* p; char* q; };
```

write a class *C* that allows the use of *x* and *p* from some *S* and *T*, much as if *x* and *p* had been members of *C*.

21. (∗1.5) Define a class *Index* to hold the index for an exponentiation function *mypow*(*double*, *Index*). Find a way to have *2*I* call *mypow*(*2*, *I*).

22. (∗2) Define a class *Imaginary* to represent imaginary numbers. Define class *Complex* based on that. Implement the fundamental arithmetic operators.