13

Templates

Your quote here.
– B. Stroustrup

Templates — a string template — instantiation — template parameters — type checking — function templates — template argument deduction — specifying template arguments — function template overloading — policy as template arguments — default template arguments — specialization — derivation and templates — member templates — conversions — source code organization — advice — exercises.

13.1 Introduction [temp.intro]

Independent concepts should be independently represented and should be combined only when needed. Where this principle is violated, you either bundle unrelated concepts together or create unnecessary dependencies. Either way, you get a less flexible set of components out of which to compose systems. Templates provide a simple way to represent a wide range of general concepts and simple ways to combine them. The resulting classes and functions can match hand-written, more-specialized code in run-time and space efficiency.

Templates provide direct support for generic programming (§2.7), that is, programming using types as parameters. The C++ template mechanism allows a type to be a parameter in the definition of a class or a function. A template depends only on the properties that it actually uses from its parameter types and does not require different types used as arguments to be explicitly related. In particular, the argument types used for a template need not be from a single inheritance hierarchy.

Here, templates are introduced with the primary focus on techniques needed for the design, implementation, and use of the standard library. The standard library requires a greater degree of generality, flexibility, and efficiency than does most software. Consequently, techniques that can be used in the design and implementation of the standard library are effective and efficient in the design of solutions to a wide variety of problems. These techniques enable an implementer to hide

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.

sophisticated implementations behind simple interfaces and to expose complexity to the user only when the user has a specific need for it. For example, ***sort(v)*** can be the interface to a variety of sort algorithms for elements of a variety of types held in a variety of containers. The sort function that is most appropriate for the particular ***v*** will be automatically chosen.

Every major standard library abstraction is represented as a template (for example, ***string***, ***ostream***, ***complex***, ***list***, and ***map***) and so are the key operations (for example, ***string*** compare, the output operator <<, ***complex*** addition, getting the next element from a ***list***, and ***sort()***). This makes the library chapters (Part 3) of this book a rich source of examples of templates and programming techniques relying on them. Consequently, this chapter concentrates on smaller examples illustrating technical aspects of templates and fundamental techniques for using them:

§13.2: The basic mechanisms for defining and using class templates
§13.3: Function templates, function overloading, and type deduction
§13.4: Template parameters used to specify policies for generic algorithms
§13.5: Multiple definitions providing alternative implementations for a template
§13.6: Derivation and templates (run-time and compile-time polymorphism)
§13.7: Source code organization

Templates were introduced in §2.7.1 and §3.8. Detailed rules for template name resolution, template syntax, etc., can be found in §C.13.

13.2 A Simple String Template [temp.string]

Consider a string of characters. A string is a class that holds characters and provides operations such as subscripting, concatenation, and comparison that we usually associate with the notion of a ''string.'' We would like to provide that behavior for many different kinds of characters. For example, strings of signed characters, of unsigned characters, of Chinese characters, of Greek characters, etc., are useful in various contexts. Thus, we want to represent the notion of ''string'' with minimal dependence on a specific kind of character. The definition of a string relies on the fact that a character can be copied, and little else. Thus, we can make a more general string type by taking the string of ***char*** from §11.12 and making the character type a parameter:

```
template<class C> class String {
    struct Srep;
    Srep *rep;
public:
    String();
    String(const C*);
    String(const String&);

    C read(int i) const;
    // ...
};
```

The ***template <class C>*** prefix specifies that a template is being declared and that a type argument ***C*** will be used in the declaration. After its introduction, ***C*** is used exactly like other type names. The scope of ***C*** extends to the end of the declaration prefixed by ***template <class C>***. Note that ***template<class C>*** says that ***C*** is a *type* name; it need not be the name of a *class*.

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.

The name of a class template followed by a type bracketed by < > is the name of a class (as defined by the template) and can be used exactly like other class names. For example:

```
String<char> cs;
String<unsigned char> us;
String<wchar_t> ws;

class Jchar {
    // Japanese character
};

String<Jchar> js;
```

Except for the special syntax of its name, ***String<char>*** works exactly as if it had been defined using the definition of class ***String*** in §11.12. Making ***String*** a template allows us to provide the facilities we had for ***String*** of ***char*** for ***String***s of any kind of character. For example, if we use the standard library ***map*** and the ***String*** template, the word-counting example from §11.8 becomes:

```
int main()      // count the occurrences of each word on input
{
    String<char> buf;
    map<String<char>,int> m;
    while (cin>>buf) m[buf]++;
    // write out result
}
```

The version for our Japanese-character type ***Jchar*** would be:

```
int main()      // count the occurrences of each word on input
{
    String<Jchar> buf;
    map<String<Jchar>,int> m;
    while (cin>>buf) m[buf]++;
    // write out result
}
```

The standard library provides the template class ***basic_string*** that is similar to the templatized ***String*** (§11.12, §20.3). In the standard library, ***string*** is defined as a synonym for ***basic_string<char>***:

```
typedef basic_string<char> string;
```

This allows us to write the word-counting program like this:

```
int main()      // count the occurrences of each word on input
{
    string buf;
    map<string,int> m;
    while (cin>>buf) m[buf]++;
    // write out result
}
```

In general, ***typedef***s are useful for shortening the long names of classes generated from templates.

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.

Also, we often prefer not to know the details of how a type is defined, and a ***typedef*** allows us to hide the fact that a type is generated from a template.

13.2.1 Defining a Template [temp.string.details]

A class generated from a class template is a perfectly ordinary class. Thus, use of a template does not imply any run-time mechanisms beyond what is used for an equivalent ''hand-written'' class. Nor does it necessarily imply any reduction in the amount of code generated.

It is usually a good idea to debug a particular class, such as ***String***, before turning it into a template such as ***String<C>***. By doing so, we handle many design problems and most of the code errors in the context of a concrete example. This kind of debugging is familiar to all programmers, and most people cope better with a concrete example than with an abstract concept. Later, we can deal with any problems that might arise from generalization without being distracted by more conventional errors. Similarly, when trying to understand a template, it is often useful to imagine its behavior for a particular type argument such as ***char*** before trying to comprehend the template in its full generality.

Members of a template class are declared and defined exactly as they would have been for a non-template class. A template member need not be defined within the template class itself. In that case, its definition must be provided somewhere else, as for non-template class members (§C.13.7). Members of a template class are themselves templates parameterized by the parameters of their template class. When such a member is defined outside its class, it must explicitly be declared a template. For example:

```
template<class C> struct String<C>::Srep {
    C* s;           // pointer to elements
    int sz;         // number of elements
    int n;          // reference count
    // ...
};

template<class C> C String<C>::read(int i) const { return rep->s[i]; }

template<class C> String<C>::String()
{
    p = new Srep(0,C());
}
```

Template parameters, such as ***C***, are parameters rather than names of types defined externally to the template. However, that doesn't affect the way we write the template code using them. Within the scope of ***String<C>***, qualification with ***<C>*** is redundant for the name of the template itself, so ***String<C>::String*** is the name for the constructor. If you prefer, you can be explicit:

```
template<class C> String<C>::String<C>()
{
    p = new Srep(0,C());
}
```

Just as there can be only one function defining a class member function in a program, there can be only one function template defining a class template member function in a program. However,

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.

overloading is a possibility for functions only (§13.3.2), while specialization (§13.5) enables us to provide alternative implementations for a template.

It is not possible to overload a class template name, so if a class template is declared in a scope, no other entity can be declared there with the same name (see also §13.5). For example:

```
template<class T> class String { /* ... */ };

class String { /* ... */ }; // error: double definition
```

A type used as a template argument must provide the interface expected by the template. For example, a type used as an argument to **String** must provide the usual copy operations (§10.4.4.1, §20.2.1). Note that there is no requirement that different arguments for the same template parameter should be related by inheritance.

13.2.2 Template Instantiation [temp.string.inst]

The process of generating a class declaration from a template class and a template argument is often called *template instantiation* (§C.13.7). Similarly, a function is generated (''instantiated'') from a template function plus a template argument. A version of a template for a particular template argument is called a *specialization*.

In general, it is the implementation's job – *not* the programmer's – to ensure that versions of a template function are generated for each set of template arguments used (§C.13.7). For example:

```
String<char> cs;

void f()
{
    String<Jchar> js;

    cs = "It´s the implementation´s job to figure out what code needs to be generated";
}
```

For this, the implementation generates declarations for **String<char>** and **String<Jchar>**, for their corresponding **Srep** types, for their destructors and default constructors, and for the assignment **String<char>::operator=(char*)**. Other member functions are not used and should not be generated. The generated classes are perfectly ordinary classes that obey all the usual rules for classes. Similarly, generated functions are ordinary functions that obey all the usual rules for functions.

Obviously, templates provide a powerful way of generating code from relatively short definitions. Consequently, a certain amount of caution is in order to avoid flooding memory with almost identical function definitions (§13.5).

13.2.3 Template Parameters [temp.param]

A template can take type parameters, parameters of ordinary types such as **int**s, and template parameters (§C.13.3). Naturally, a template can take several parameters. For example:

```
template<class T, T def_val> class Cont { /* ... */ };
```

As shown, a template parameter can be used in the definition of subsequent template parameters.

Integer arguments come in handy for supplying sizes and limits. For example:

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.

```
template<class T, int i> class Buffer {
        T v[i];
        int sz;
public:
        Buffer() : sz(i) {}
        // ...
};

Buffer<char,127> cbuf;
Buffer<Record,8> rbuf;
```

Simple and constrained containers such as ***Buffer*** can be important where run-time efficiency and compactness are paramount (thus preventing the use of a more general ***string*** or ***vector***). Passing a size as a template argument allows ***Buffer***'s implementer to avoid free store use. Another example is the ***Range*** type in §25.6.1.

A template argument can be a constant expression (§C.5), the address of an object or function with external linkage (§9.2), or a non-overloaded pointer to member (§15.5). A pointer used as a template argument must be of the form *&of*, where ***of*** is the name of an object or a function, or of the form ***f***, where ***f*** is the name of a function. A pointer to member must be of the form *&X::of*, where ***of*** is the name of an member. In particular, a string literal is *not* acceptable as a template argument.

An integer template argument must be a constant:

```
void f(int i)
{
        Buffer<int,i> bx;       // error: constant expression expected
}
```

Conversely, a non-type template parameter is a constant within the template so that an attempt to change the value of a parameter is an error.

13.2.4 Type Equivalence [temp.equiv]

Given a template, we can generate types by supplying template arguments. For example:

```
String<char> s1;
String<unsigned char> s2;
String<int> s3;

typedef unsigned char Uchar;
String<Uchar> s4;
String<char> s5;

Buffer<String<char>,10> b1;
Buffer<char,10> b2;
Buffer<char,20-10> b3;
```

When using the same set of template arguments for a template, we always refer to the same generated type. However, what does ''the same'' mean in this context? As usual, ***typedef***s do not introduce new types, so ***String<Uchar>*** is the same type as ***String<unsigned char>***. Conversely,

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.

because *char* and *unsigned char* are different types (§4.3), *String<char>* and *String<unsigned char>* are different types.

The compiler can evaluate constant expressions (§C.5), so *Buffer<char,20-10>* is recognized to be the same type as *Buffer<char,10>*.

13.2.5 Type Checking [temp.check]

A template is defined and then later used in combination with a set of template arguments. When the template is defined, the definition is checked for syntax errors and possibly also for other errors that can be detected in isolation from a particular set of template arguments. For example:

```
template<class T> class List {
    struct Link {
        Link* pre;
        Link* suc;
        T val;
        Link(Link* p, Link* s, const T& v) :pre(p), suc(s), val(v) { }
    }           // syntax error: missing semicolon
    Link* head;
public:
    List() : head(7) { }        // error: pointer initialized with int
    List(const T& t) : head(new Link(0,o,t)) { }        // error: undefined identifier 'o'
    // ...
    void print_all() { for (Link* p = head; p; p=p->suc) cout << p->val << '\n'; }
};
```

A compiler can catch simple semantic errors at the point of definition or later at the point of use. Users generally prefer early detection, but not all ''simple'' errors are easy to detect. Here, I made three ''mistakes.'' Independently of what the template parameter is, a pointer *T** cannot be initialized by the integer *7*. Similarly, the identifier *o* (a mistyped *0*, of course) cannot be an argument to *List<T>::Link*'s constructor because there is no such name in scope.

A name used in a template definition must either be in scope or in some reasonably obvious way depend on a template parameter (§C.13.8.1). The most common and obvious way of depending on a template parameter *T* is to use a member of a *T* or to take an argument of type *T*. In *List<T>::print_all()*, *cout<<p->val* is a slightly more subtle example.

Errors that relate to the use of template parameters cannot be detected until the template is used. For example:

```
class Rec { /* ... */ };

void f(List<int>& li, List<Rec>& lr)
{
    li.print_all();
    lr.print_all();
}
```

The *li.print_all()* checks out fine, but *lr.print_all()* gives a type error because there is no << output operator defined for *Rec*. The earliest that errors relating to a template parameter can be detected is at the first point of use of the template for a particular template argument. That point is

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.

usually called the *first point of instantiation*, or simply the *point of instantiation* (see §C.13.7). The implementation is allowed to postpone this checking until the program is linked. If we had only a declaration of ***print_all()*** available in this translation unit, rather than its definition, the implementation might have had to delay type checking (see §13.7). Independently of when checking is done, the same set of rules is checked. Again, users prefer early checking. It is possible to express constraints on template arguments in terms of member functions (see §13.9[16]).

13.3 Function Templates [temp.fct]

For most people, the first and most obvious use of templates is to define and use container classes such as ***basic_string*** (§20.3), ***vector*** (§16.3), ***list*** (§17.2.2), and ***map*** (§17.4.1). Soon after, the need for template functions arises. Sorting an array is a simple example:

```
template<class T> void sort(vector<T>&);        // declaration

void f(vector<int>& vi, vector<string>& vs)
{
    sort(vi);  // sort(vector<int>&);
    sort(vs);  // sort(vector<string>&);
}
```

When a template function is called, the types of the function arguments determine which version of the template is used; that is, the template arguments are deduced from the function arguments (§13.3.1).

Naturally, the template function must be defined somewhere (§C.13.7):

```
template<class T> void sort(vector<T>& v)       // definition
    // Shell sort (Knuth, Vol. 3, pg. 84).
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (v[j+gap]<v[j]) {      // swap v[j] and v[j+gap]
                    T temp = v[j];
                    v[j] = v[j+gap];
                    v[j+gap] = temp;
                }
}
```

Please compare this definition to the ***sort()*** defined in (§7.7). This templatized version is cleaner and shorter because it can rely on more information about the type of the elements it sorts. Most likely, it is also faster because it doesn't rely on a pointer to function for the comparison. This implies that no indirect function calls are needed and that inlining of a simple < is easy.

A further simplification is to use the standard library template ***swap()*** (§18.6.8) to reduce the action to its natural form:

```
if (v[j+gap]<v[j]) swap(v[j],v[j+gap]);
```

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.

This does not introduce any new overheads.

In this example, operator < is used for comparison. However, not every type has a < operator. This limits the use of this version of *sort*( ), but the limitation is easily avoided (see §13.4).

13.3.1 Function Template Arguments [temp.deduce]

Function templates are essential for writing generic algorithms to be applied to a wide variety of container types (§2.7.2, §3.8, Chapter 18). The ability to deduce the template arguments for a call from the function arguments is crucial.

A compiler can deduce type and non-type arguments from a call, provided the function argument list uniquely identifies the set of template arguments (§C.13.4). For example:

```
template<class T, int i> T lookup(Buffer<T,i>& b, const char* p);

class Record {
    const char[12];
    // ...
};

Record f(Buffer<Record,128>& buf, const char* p)
{
    return lookup(buf,p); // use the lookup() where T is Record and i is 128
}
```

Here, *T* is deduced to be *Record* and *i* is deduced to be *128*.

Note that class template parameters are never deduced. The reason is that the flexibility provided by several constructors for a class would make such deduction impossible in many cases and obscure in many more. Specialization provides a mechanism for implicitly choosing between different implementations of a class (§13.5). If we need to create an object of a deduced type, we can often do that by calling a function to do the creation; see *make_pair*( ) in §17.4.1.2.

If a template argument cannot be deduced from the template function arguments (§C.13.4), we must specify it explicitly. This is done in the same way template arguments are explicitly specified for a template class. For example:

```
template<class T> class vector { /* ... */ };
template<class T> T* create(); // make a T and return a pointer to it

void f()
{
    vector<int> v;              // class, template argument 'int'
    int* p = create<int>();     // function, template argument 'int'
}
```

One common use of explicit specification is to provide a return type for a template function:

```
template<class T, class U> T implicit_cast(U u) { return u; }

void g(int i)
{
    implicit_cast(i);           // error: can't deduce T
    implicit_cast<double>(i);   // T is double; U is int
```

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.

```
    implicit_cast<char,double>(i);  // T is char; U is double
    implicit_cast<char*,int>(i);    // T is char*; U is int; error: cannot convert int to char*
}
```

As with default function arguments (§7.5), only trailing arguments can be left out of a list of explicit template arguments.

Explicit specification of template arguments allows the definition of families of conversion functions and object creation functions (§13.3.2, §C.13.1, §C.13.5). An explicit version of the implicit conversions (§C.6), such as ***implicit_cast***( ), is frequently useful. The syntax for ***dynamic_cast***, ***static_cast***, etc., (§6.2.7, §15.4.1) matches the explicitly qualified template function syntax. However, the built-in type conversion operators supply operations that cannot be expressed by other language features.

13.3.2 Function Template Overloading [temp.over]

One can declare several function templates with the same name and even declare a combination of function templates and ordinary functions with the same name. When an overloaded function is called, overload resolution is necessary to find the right function or template function to invoke. For example:

```
template<class T> T sqrt(T);
template<class T> complex<T> sqrt(complex<T>);
double sqrt(double);

void f(complex<double> z)
{
    sqrt(2);        // sqrt<int>(int)
    sqrt(2.0);      // sqrt(double)
    sqrt(z);        // sqrt<double>(complex<double>)
}
```

In the same way that a template function is a generalization of the notion of a function, the rules for resolution in the presence of function templates are generalizations of the function overload resolution rules. Basically, for each template we find the specialization that is best for the set of function arguments. Then, we apply the usual function overload resolution rules to these specializations and all ordinary functions:

[1] Find the set of function template specializations (§13.2.2) that will take part in overload resolution. Do this by considering each function template and deciding which template arguments, if any, would be used if no other function templates or functions of the same name were in scope. For the call ***sqrt***(*z*), this makes ***sqrt<double>***(***complex<double>***) and ***sqrt< complex<double> >***(***complex<double>***) candidates.

[2] If two template functions can be called and one is more specialized than the other (§13.5.1), consider only the most specialized template function in the following steps. For the call ***sqrt***(*z*), this means that ***sqrt<double>***(***complex<double>***) is preferred over ***sqrt< complex<double> >***(***complex<double>***): any call that matches ***sqrt<T>***(***complex<T>***) also matches ***sqrt<T>***(*T*).

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.

[3] Do overload resolution for this set of functions, plus any ordinary functions as for ordinary functions (§7.4). If a template function argument has been determined by template argument deduction (§13.3.1), that argument cannot also have promotions, standard conversions, or user-defined conversions applied. For ***sqrt(2)***, ***sqrt<int>(int)*** is an exact match, so it is preferred over ***sqrt(double)***.

[4] If a function and a specialization are equally good matches, the function is preferred. Consequently, ***sqrt(double)*** is preferred over ***sqrt<double>(double)*** for ***sqrt(2.0)***.

[5] If no match is found, the call is an error. If we end up with two or more equally good matches, the call is ambiguous and is an error.

For example:

```
template<class T> T max(T,T);

const int s = 7;

void k()
{
    max(1,2);           // max<int>(1,2)
    max('a','b');       // max<char>('a','b')
    max(2.7,4.9);       // max<double>(2.7,4.9)
    max(s,7);           // max<int>(int(s),7) (trivial conversion used)

    max('a',1);         // error: ambiguous (no standard conversion)
    max(2.7,4);         // error: ambiguous (no standard conversion)
}
```

We could resolve the two ambiguities either by explicit qualification:

```
void f()
{
    max<int>('a',1);        // max<int>(int('a'),1)
    max<double>(2.7,4);     // max<double>(2.7,double(4))
}
```

or by adding suitable declarations:

```
inline int max(int i, int j) { return max<int>(i,j); }
inline double max(int i, double d) { return max<double>(i,d); }
inline double max(double d, int i) { return max<double>(d,i); }
inline double max(double d1, double d2) { return max<double>(d1,d2); }

void g()
{
    max('a',1);   // max(int('a'),1)
    max(2.7,4);   // max(2.7,double(4))
}
```

For ordinary functions, ordinary overloading rules (§7.4) apply, and the use of ***inline*** ensures that no extra overhead is imposed.

The definition of ***max()*** is trivial, so we could have written it explicitly. However, using a specialization of the template is an easy and general way of defining such resolution functions.

The overload resolution rules ensure that template functions interact properly with inheritance:

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.

```
template<class T> class B { /* ... */ };
template<class T> class D : public B<T> { /* ... */ };

template<class T> void f(B<T>*);

void g(B<int>* pb, D<int>* pd)
{
    f(pb);    // f<int>(pb)
    f(pd);    // f<int>(static_cast<B<int>*>(pd)); standard conversion D<int>* to B<int>* used
}
```

In this example, the template function ***f( )*** accepts a ***B<T>**** for any type ***T***. We have an argument of type ***D<int>****, so the compiler easily deduces that by choosing ***T*** to be ***int***, the call can be uniquely resolved to a call of ***f(B<int>*)***.

A function argument that is not involved in the deduction of a template parameter is treated exactly as an argument of a non-template function. In particular, the usual conversion rules hold. Consider:

```
template<class C> int get_nth(C& p, int n);    // get n-th element
```

This function presumably returns the value of the n-th element of a container of type ***C***. Because ***C*** has to be deduced from an actual argument of ***get_nth( )*** in a call, conversions are not applicable to the first argument. However, the second argument is perfectly ordinary, so the full range of possible conversions is considered. For example:

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

void f(vector<int>& v, short s, Index i)
{
    int i1 = get_nth(v,2);    // exact match
    int i2 = get_nth(v,s);    // standard conversion: short to int
    int i3 = get_nth(v,i);    // user-defined conversion: Index to int
}
```

13.4 Using Template Arguments to Specify Policy [temp.policy]

Consider how to sort strings. Three concepts are involved: the string, the element type, and the criteria used by the sort algorithm for comparing string elements.

We can't hardwire the sorting criteria into the container because the container can't (in general) impose its needs on the element types. We can't hardwire the sorting criteria into the element type because there are many different ways of sorting elements.

Consequently, the sorting criteria are built neither into the container nor into the element type. Instead, the criteria must be supplied when a specific operation needs to be performed. For example, if I have strings of characters representing names of Swedes, what collating criteria would I

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.

like to use for a comparison? Two different collating sequences (numerical orderings of the characters) are commonly used for sorting Swedish names. Naturally, neither a general string type nor a general sort algorithm should know about the conventions for sorting names in Sweden. Therefore, any general solution requires that the sorting algorithm be expressed in general terms that can be defined not just for a specific type but also for a specific use of a specific type. For example, let us generalize the standard C library function ***strcmp()*** for ***String***s of any type ***T*** (§13.2):

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
        if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
    return str1.length()-str2.length();
}
```

If someone wants ***compare()*** to ignore case, to reflect locale, etc., that can be done by defining suitable ***C::eq()*** and ***C::lt()***. This allows any (comparison, sorting, etc.) algorithm that can be described in terms of the operations supplied by the ''***C***-operations'' and the container to be expressed. For example:

```
template<class T> class Cmp { // normal, default compare
public:
    static int eq(T a, T b) { return a==b; }
    static int lt(T a, T b) { return a<b; }
};

class Literate { // compare Swedish names according to literary conventions
public:
    static int eq(char a, char b) { return a==b; }
    static int lt(char,char); // a table lookup based on character value (§13.9[14])
};
```

We can now choose the rules for comparison by explicit specification of the template arguments:

```
void f(String<char> swede1, String<char> swede2)
{
    compare< char,Cmp<char> >(swede1,swede2);
    compare< char,Literate >(swede1,swede2);
}
```

Passing the comparison operations as a template parameter has two significant benefits compared to alternatives such as passing pointers to functions. Several operations can be passed as a single argument with no run-time cost. In addition, the comparison operators ***eq()*** and ***lt()*** are trivial to inline, whereas inlining a call through a pointer to function requires exceptional attention from a compiler.

Naturally, comparison operations can be provided for user-defined types as well as built-in types. This is essential to allow general algorithms to be applied to types with nontrivial comparison criteria (see §18.4).

Each class generated from a class template gets a copy of each ***static*** member of the class template (see §C.13.1).

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.

13.4.1 Default Template Parameters [temp.default]

Explicitly specifying the comparison criteria for each call is tedious. Fortunately, it is easy to pick a default so that only uncommon comparison criteria have to be explicitly specified. This can be implemented through overloading:

```
template<class T, class C>
int compare(const String<T>& str1, const String<T>& str2); // compare using C

template<class T>
int compare(const String<T>& str1, const String<T>& str2); // compare using Cmp<T>
```

Alternatively, we can supply the normal convention as a default template argument:

```
template<class T, class C = Cmp<T> >
int compare(const String<T>& str1, const String<T>& str2)
{
    for(int i=0; i<str1.length() && i< str2.length(); i++)
        if (!C::eq(str1[i],str2[i])) return C::lt(str1[i],str2[i]) ? -1 : 1;
    return str1.length()-str2.length();
}
```

Given that, we can write:

```
void f(String<char> swede1, String<char> swede2)
{
    compare(swede1,swede2);                 // use Cmp<char>
    compare<char,Literate>(swede1,swede2);  // use Literate
}
```

A less esoteric example (for non-Swedes) is comparing with and without taking case into account:

```
class No_case { /* ... */ };

void f(String<char> s1, String<char> s2)
{
    compare(s1,s2);                 // case sensitive
    compare<char,No_case>(s1,s2);   // not sensitive to case
}
```

The technique of supplying a policy through a template argument and then defaulting that argument to supply the most common policy is widely used in the standard library (e.g., §18.4). Curiously enough, it is not used for ***basic_string*** (§13.2, Chapter 20) comparisons. Template parameters used to express policies are often called ''traits.'' For example, the standard library string relies on ***char_traits*** (§20.2.1), the standard algorithms on iterator traits (§19.2.2), and the standard library containers on ***allocators*** (§19.4).

The semantic checking of a default argument for a template parameter is done if and (only) when that default argument is actually used. In particular, as long as we refrain from using the default template argument ***Cmp<T>*** we can ***compare()*** strings of a type ***X*** for which ***Cmp<X>*** wouldn't compile (say, because < wasn't defined for an ***X***). This point is crucial in the design of the standard containers, which rely on a template argument to specify default values (§16.3.4).

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.

13.5 Specialization [temp.special]

By default, a template gives a single definition to be used for every template argument (or combination of template arguments) that a user can think of. This doesn't always make sense for someone writing a template. I might want to say, ''if the template argument is a pointer, use this implementation; if it is not, use that implementation'' or ''give an error unless the template argument is a pointer derived from class *My_base*.'' Many such design concerns can be addressed by providing alternative definitions of the template and having the compiler choose between them based on the template arguments provided where they are used. Such alternative definitions of a template are called *user-defined specializations*, or simply, *user specializations*.

Consider likely uses of a *Vector* template:

```
template<class T> class Vector {      // general vector type
        T* v;
        int sz;
public:
        Vector();
        Vector(int);

        T& elem(int i) { return v[i]; }
        T& operator[](int i);

        void swap(Vector&);
        // ...
};

Vector<int> vi;
Vector<Shape*> vps;
Vector<string> vs;
Vector<char*> vpc;
Vector<Node*> vpn;
```

Most *Vector*s will be *Vector*s of some pointer type. There are several reasons for this, but the primary reason is that to preserve run-time polymorphic behavior, we must use pointers (§2.5.4, §12.2.6). That is, anyone who practices object-oriented programming and also uses type-safe containers (such as the standard library containers) will end up with a lot of containers of pointers.

The default behavior of most C++ implementations is to replicate the code for template functions. This is good for run-time performance, but unless care is taken it leads to code bloat in critical cases such as the *Vector* example.

Fortunately, there is an obvious solution. Containers of pointers can share a single implementation. This can be expressed through specialization. First, we define a version (a specialization) of *Vector* for pointers to *void*:

```
template<> class Vector<void*> {
        void** p;
        // ...
        void*& operator[](int i);
};
```

This specialization can then be used as the common implementation for all *Vector*s of pointers.

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.

The *template<>* prefix says that this is a specialization that can be specified without a template parameter. The template arguments for which the specialization is to be used are specified in <> brackets after the name. That is, the *<void*>* says that this definition is to be used as the implementation of every *Vector* for which *T* is void* .

The *Vector<void*>* is a complete specialization. That is, there is no template parameter to specify or deduce when we use the specialization; *Vector<void*>* is used for *Vector*s declared like this:

```
Vector<void*> vpv;
```

To define a specialization that is used for every *Vector* of pointers and only for *Vector*s of pointers, we need a *partial specialization*:

```
template<class T> class Vector<T*> : private Vector<void*> {
public:
    typedef Vector<void*> Base;

    Vector() : Base() {}
    explicit Vector(int i) : Base(i) {}

    T*& elem(int i) { return static_cast<T*&>(Base::elem(i)); }
    T*& operator[](int i) { return static_cast<T*&>(Base::operator[](i)); }

    // ...
};
```

The specialization pattern *<T*>* after the name says that this specialization is to be used for every pointer type; that is, this definition is to be used for every *Vector* with a template argument that can be expressed as *T**. For example:

```
Vector<Shape*> vps; // <T*> is <Shape*> so T is Shape
Vector<int**> vppi;  // <T*> is <int**> so T is int*
```

Note that when a partial specialization is used, a template parameter is deduced from the specialization pattern; the template parameter is not simply the actual template argument. In particular, for *Vector<Shape*>*, *T* is *Shape* and not *Shape**.

Given this partial specialization of *Vector*, we have a shared implementation for all *Vector*s of pointers. The *Vector<T*>* class is simply an interface to *void** implemented exclusively through derivation and inline expansion.

It is important that this refinement of the implementation of *Vector* is achieved without affecting the interface presented to users. Specialization is a way of specifying alternative implementations for different uses of a common interface. Naturally, we could have given the general *Vector* and the *Vector* of pointers different names. However, when I tried that, many people who should have known better forgot to use the pointer classes and found their code much larger than expected. In this case, it is much better to hide the crucial implementation details behind a common interface.

This technique proved successful in curbing code bloat in real use. People who do not use a technique like this (in C++ or in other languages with similar facilities for type parameterization) have found that replicated code can cost megabytes of code space even in moderately-sized programs. By eliminating the time needed to compile those additional versions of the vector operations, this technique can also cut compile and link times dramatically. Using a single specialization

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.

to implement all lists of pointers is an example of the general technique of minimizing code bloat by maximizing the amount of shared code.

The general template must be declared before any specialization. For example:

```
template<class T> class List<T*> { /* ... */ };
template<class T> class List { /* ... */ }; // error: general template after specialization
```

The critical information supplied by the general template is the set of template parameters that the user must supply to use it or any of its specializations. Consequently, a declaration of the general case is sufficient to allow the declaration or definition of a specialization:

```
template<class T> class List;
template<class T> class List<T*> { /* ... */ };
```

If used, the general template needs to be defined somewhere (§13.7).

If a user specializes a template somewhere, that specialization must be in scope for every use of the template with the type for which it was specialized. For example:

```
template<class T> class List { /* ... */ };
List<int*> li;
template<class T> class List<T*> { /* ... */ }; // error
```

Here, ***List*** was specialized for ***int**** after ***List<int*>*** had been used.

All specializations of a template must be declared in the same namespace as the template itself. If used, a specialization that is explicitly declared (as opposed to generated from a more general template) must also be explicitly defined somewhere (§13.7). In other words, explicitly specializing a template implies that no definition is generated for that specialization.

13.5.1 Order of Specializations [temp.special.order]

One specialization is *more specialized* than another if every argument list that matches its specialization pattern also matches the other, but not vice versa. For example:

```
template<class T> class Vector;          // general
template<class T> class Vector<T*>;      // specialized for any pointer
template<> class Vector<void*>;          // specialized for void*
```

Every type can be used as a template argument for the most general ***Vector***, but only pointers can be used for ***Vector<T*>*** and only ***void****s can be used for ***Vector<void*>***.

The most specialized version will be preferred over the others in declarations of objects, pointers, etc., (§13.5) and in overload resolution (§13.3.2).

A specialization pattern can be specified in terms of types composed using the constructs allowed for template parameter deduction (§13.3.1).

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.

13.5.2 Template Function Specialization [temp.special.fct]

Naturally, specialization is also useful for template functions. Consider the Shell sort from §7.7 and §13.3. It compares elements using < and swaps elements using detailed code. A better definition would be:

```
template<class T> bool less(T a, T b) { return a<b; }

template<class T> void sort(Vector<T>& v)
{
    const size_t n = v.size();

    for (int gap=n/2; 0<gap; gap/=2)
        for (int i=gap; i<n; i++)
            for (int j=i-gap; 0<=j; j-=gap)
                if (less(v[j+gap],v[j])) swap(v[j],v[j+gap]);
}
```

This does not improve the algorithm itself, but it allows improvements to its implementation. As written, ***sort()*** will not sort a ***Vector<char*>*** correctly because < will compare the two ***char****s. That is, it will compare the addresses of the first ***char*** in each string. Instead, we would like it to compare the characters pointed to. A simple specialization of ***less()*** for ***const char**** will take care of that:

```
template<> bool less<const char*>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

As for classes (§13.5), the ***template<>*** prefix says that this is a specialization that can be specified without a template parameter. The ***<const char*>*** after the template function name means that this specialization is to be used in cases where the template argument is ***const char****. Because the template argument can be deduced from the function argument list, we need not specify it explicitly. So, we could simplify the definition of the specialization:

```
template<> bool less<>(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

Given the ***template<>*** prefix, the second empty <> is redundant, so we would typically simply write:

```
template<> bool less(const char* a, const char* b)
{
    return strcmp(a,b)<0;
}
```

I prefer this shorter form of declaration.

Consider the obvious definition of ***swap()***:

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.

```
template<class T> void swap(T& x, T& y)
{
    T t = x;        // copy x to temporary
    x = y;          // copy y to x
    y = t;          // copy temporary to y
}
```

This is rather inefficient when invoked for ***Vector***s of ***Vector***s; it swaps ***Vector***s by copying all elements. This problem can also be solved by appropriate specialization. A ***Vector*** object will itself hold only sufficient data to give indirect access to the elements (like ***string***; §11.12, §13.2). Thus, a swap can be done by swapping those representations. To be able to manipulate that representation, I provided ***Vector*** with a member function ***swap()*** (§13.5):

```
template<class T> void Vector<T>::swap(Vector & a)      // swap representations
{
    swap(v,a.v);
    swap(sz,a.sz);

}
```

This member ***swap()*** can now be used to define a specialization of the general ***swap()***:

```
template<class T> void swap(Vector<T>& a, Vector<T>& b)
{
    a.swap(b);
}
```

These specializations of ***less()*** and ***swap()*** are used in the standard library (§16.3.9, §20.3.16). In addition, they are examples of widely applicable techniques. Specialization is useful when there is a more efficient alternative to a general algorithm for a set of template arguments (here, ***swap()***). In addition, specialization comes in handy when an irregularity of an argument type causes the general algorithm to give an undesired result (here, ***less()***). These ''irregular types'' are often the built-in pointer and array types.

13.6 Derivation and Templates [temp.derive]

Templates and derivation are mechanisms for building new types out of existing ones, and generally for writing useful code that exploits various forms of commonality. As shown in §3.7.1, §3.8.5, and §13.5, combinations of the two mechanisms are the basis for many useful techniques.

Deriving a template class from a non-template class is a way of providing a common implementation for a set of templates. The list from §13.5 is a good example of this:

```
template<class T> class list<T*> : private list<void*> { /* ... */ };
```

Another way of looking at such examples is that a template is used to provide an elegant and type-safe interface to an otherwise unsafe and inconvenient-to-use facility.

Naturally, it is often useful to derive one template class from another. One use of a base class is as a building block in the implementation of further classes. If the data or operations in such a base

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.

class depend on a template parameter of a derived class, the base itself must be parameterized; ***Vec*** from §3.7.1 is an example of this:

```
template<class T> class vector { /* ... */ };
template<class T> class Vec : public vector<T> { /* ... */ };
```

The overload resolution rules for template functions ensure that functions work ''correctly'' for such derived types (§13.3.2).

Having the same template parameter for the base and derived class is the most common case, but it is not a requirement. Interesting, although less frequently used, techniques rely on passing the derived type itself to the base class. For example:

```
template <class C> class Basic_ops { // basic operators on containers
    bool operator==(const C&) const;    // compare all elements
    bool operator!=(const C&) const;
    // ...
};

template<class T> class Math_container : public Basic_ops< Math_container<T> > {
public:
    size_t size() const;
    T& operator[](size_t);
    // ...
};
```

This allows the definition of the basic operations on containers to be separate from the definition of the containers themselves and defined once only. However, the definition of operations such as `==` and `!=` must be expressed in terms of both the container and its elements, so the base class needs to be passed to the container template.

Assuming that a ***Math_container*** is similar to a traditional vector, the definitions of a ***Basic_ops*** member would look something like this:

```
template <class C> bool Basic_ops<C>::operator==(const C& a) const
{
    if (size() != a.size()) return false;
    for (int i = 0; i<size(); ++i)
        if ((*this)[i] != a[i]) return false;
    return true;
}
```

An alternative technique for keeping the containers and operations separate would be to combine them from template arguments rather than use derivation:

```
template<class T, class C> class Mcontainer {
    C elements;
public:
    // ...
    T& operator[](size_t i) { return elements[i]; }
```

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.

```
    friend bool operator==(const Mcontainer&, const Mcontainer&); // compare elements
    friend bool operator!=(const Mcontainer&, const Mcontainer&);
    // ...
};

template<class T> class My_array { /* ... */ };

Mcontainer< double,My_array<double> > mc;
```

A class generated from a class template is a perfectly ordinary class. Consequently, it can have ***friend*** functions (§C.13.2). In this case, I used ***friend***s to achieve the conventional symmetric argument style for `==` and `!=` (§11.3.2). One might also consider passing a template rather than a container as the ***C*** argument in such cases (§13.2.3).

13.6.1 Parameterization and Inheritance [temp.inherit]

A template parameterizes the definition of a type or a function with another type. Code implementing the template is identical for all parameter types, as is most code using the template. An abstract class defines an interface. Much code for different implementations of the abstract class can be shared in class hierarchies, and most code using the abstract class doesn't depend on its implementation. From a design perspective, the two approaches are so close that they deserve a common name. Since both allow an algorithm to be expressed once and applied to a variety of types, people sometimes refer to both as ***polymorphic***. To distinguish them, what virtual functions provide is called *run-time polymorphism*, and what templates offer is called *compile-time polymorphism* or *parametric polymorphism*.

So when do we choose to use a template and when do we rely on an abstract class? In either case, we manipulate objects that share a common set of operations. If no hierarchical relationship is required between these objects, they are best used as template arguments. If the actual types of these objects cannot be known at compile-time, they are best represented as classes derived from a common abstract class. If run-time efficiency is at a premium, that is, if inlining of operations is essential, a template should be used. This issue is discussed in greater detail in §24.4.1.

13.6.2 Member Templates [temp.member]

A class or a class template can have members that are themselves templates. For example:

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    template<class T>
        complex(const complex<T>& c) : re(c.re), im(c.im) { }
    // ...
};

complex<float> cf(0,0);
complex<double> cd = cf;  // ok: uses float to double conversion
```

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.

```
class Quad {
    // no conversion to int
};

complex<Quad> cq;
complex<int> ci = cq;          // error: no Quad to int conversion
```

In other words, you can construct a ***complex<T1>*** from a ***complex<T2>*** if and only if you can initialize a ***T1*** by a ***T2***. That seems reasonable.

Unfortunately, C++ accepts some unreasonable conversions between built-in types, such as from ***double*** to ***int***. Truncation problems could be caught at run time using a checked conversion in the style of ***implicit_cast*** (§13.3.1) and ***checked*** (§C.6.2.6):

```
template<class Scalar> class complex {
    Scalar re, im;
public:
    complex() : re(0), im(0) { }
    complex(const complex<Scalar>& c) : re(c.re), im(c.im) { }

    template<class T2> complex(const complex<T2>& c)
        : re(checked_cast<Scalar>(c.real())), im(checked_cast<Scalar>(c.imag())) { }
    // ...
};
```

For completeness, I added a default constructor and a copy constructor. Curiously enough, a template constructor is never used to generate a copy constructor, so without the explicitly declared copy constructor, a default copy constructor would have been generated. In that case, that generated copy constructor would have been identical to the one I explicitly specified.

A member template cannot be ***virtual***. For example:

```
class Shape {
    // ...
    template<class T> virtual bool intersect(const T&) const =0; // error: virtual template
};
```

This must be illegal. If it were allowed, the traditional virtual function table technique for implementing virtual functions (§2.5.5) could not be used. The linker would have to add a new entry to the virtual table for class ***Shape*** each time someone called ***intersect()*** with a new argument type.

13.6.3 Inheritance Relationships [temp.rel.inheritance]

A class template is usefully understood as a specification of how particular types are to be created. In other words, the template implementation is a mechanism that generates types when needed based on the user's specification. Consequently, a class template is sometimes called a *type generator*.

As far as the C++ language rules are concerned, there is no relationship between two classes generated from a single class template. For example:

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

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.

Given these declarations, people sometimes try to treat a ***set<Circle*>*** as a ***set<Shape*>***. This is a serious logical error based on a flawed argument: ''A ***Circle*** is a ***Shape***, so a set of ***Circles*** is also a set of ***Shapes***; therefore, I should be able to use a set of ***Circles*** as a set of ***Shapes***.'' The ''therefore'' part of this argument doesn't hold. The reason is that a set of ***Circle***s guarantees that the member of the set are ***Circles***; a set of ***Shape***s does not provide that guarantee. For example:

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

void f(set<Shape*>& s)
{
    // ...
    s.insert(new Triangle());
    // ...
}

void g(set<Circle*>& s)
{
    f(s); // error, type mismatch: s is a set<Circle*>, not a set<Shape*>
}
```

This won't compile because there is no built-in conversion from ***set<Circle*>&*** to ***set<Shape*>&***. Nor should there be. The guarantee that the members of a ***set<Circle*>*** are ***Circle***s allows us to safely and efficiently apply ***Circle***-specific operations, such as determining the radius, to members of the set. If we allowed a ***set<Circle*>*** to be treated as a ***set<Shape*>***, we could no longer maintain that guarantee. For example, ***f( )*** inserts a ***Triangle**** into its ***set<Shape*>*** argument. If the ***set<Shape*>*** could have been a ***set<Circle*>***, the fundamental guarantee that a ***set<Circle*>*** contains ***Circle****s only would have been violated.

13.6.3.1 Template Conversions [temp.mem.temp]

The example in the previous section demonstrates that there cannot be any *default* relationship between classes generated from the same templates. However, for some templates we would like to express such a relationship. For example, when we define a pointer template, we would like to reflect inheritance relationships among the objects pointed to. Member templates (§13.6.2) allow us to specify many such relationships where desired. Consider:

```
template<class T> class Ptr {    // pointer to T
    T* p;
public:
    Ptr(T*);
    template<class T2> operator Ptr<T2> (); // convert Ptr<T> to Ptr<T2>
    // ...
};
```

We would like to define the conversion operators to provide the inheritance relationships we are accustomed to for built-in pointers for these user-defined ***Ptr***s. For example:

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.

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc;        // should work
    Ptr<Circle> pc2 = ps;      // should give error
}
```

We want to allow the first initialization if and only if ***Shape*** really is a direct or indirect public base class of ***Circle***. In general, we need to define the conversion operator so that the ***Ptr<T>*** to ***Ptr<T2>*** conversion is accepted if and only if a ***T**** can be assigned to a ***T2****. That can be done like this:

```
template<class T>
    template<class T2>
        Ptr<T>::operator Ptr<T2>() { return Ptr<T2>(p); }
```

The return statement will compile if and only if ***p*** (which is a ***T****) can be an argument to the ***Ptr<T2>(T2*)*** constructor. Therefore, if ***T**** can be implicitly converted into a ***T2****, the ***Ptr<T>*** to ***Ptr<T2>*** conversion will work. For example

```
void f(Ptr<Circle> pc)
{
    Ptr<Shape> ps = pc;        // ok: can convert Circle* to Shape*
    Ptr<Circle> pc2 = ps;      // error: cannot convert Shape* to Circle*
}
```

Be careful to define logically meaningful conversions only.

Note that the template parameter lists of a template and its template member cannot be combined. For example:

```
template<class T, class T2>    // error
    Ptr<T>::operator Ptr<T2>() { return Ptr<T2>(p); }
```

13.7 Source Code Organization [temp.source]

There are two obvious ways of organizing code using templates:

[1] Include template definitions before their use in a translation unit.

[2] Include template declarations (only) before their use in a translation unit, and compile their definitions separately.

In addition, template functions are sometimes first declared, then used, and finally defined in a single translation unit.

To see the differences between the two main approaches, consider a simple template:

```
#include<iostream>

template<class T> void out(const T& t) { std::cerr << t; }
```

We could call this ***out.c*** and ***#include*** it wherever ***out()*** was needed. For example:

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.

```
// user1.c:
    #include "out.c"
    // use out()

// user2.c:
    #include "out.c"
    // use out()
```

That is, the definition of `out()` and all declarations it depends on are `#include`d in several different compilation units. It is up to the compiler to generate code when needed (only) and to optimize the process of reading redundant definitions. This strategy treats template functions the same way as inline functions.

One obvious problem with this is that everything on which the definition of `out()` depends is added to each file using `out()`, thus increasing the amount of information that the compiler must process. Another problem is that users may accidentally come to depend on declarations included only for the benefit of the definition of `out()`. This danger can be minimized by using namespaces, by avoiding macros, and generally by reducing the amount of information included.

The separate compilation strategy is the logical conclusion of this line of thinking: if the template definition isn't included in the user code, none of its dependencies can affect that code. Thus we split the original `out.c` into two files:

```
// out.h:
    template<class T> void out(const T& t);

// out.c:
    #include<iostream>
    #include "out.h"

    export template<class T> void out(const T& t) { std::cerr << t; }
```

The file `out.c` now holds all of the information needed to define `out()`, and `out.h` holds only what is needed to call it. A user `#include`s only the declaration (the interface):

```
// user1.c:
    #include "out.h"
    // use out()

// user2.c:
    #include "out.h"
    // use out()
```

This strategy treats template functions the same way it does non-inline functions. The definition (in `out.c`) is compiled separately, and it is up to the implementation to find the definition of `out()` when needed. This strategy also puts a burden on the implementation. Instead of having to filter out redundant copies of a template definition, the implementation must find the unique definition when needed.

Note that to be accessible from other compilation units, a template definition must be explicitly declared *export* (§9.2.3). This can be done by adding *export* to the definition or to a preceding declaration. Otherwise, the definition must be in scope wherever the template is used.

Which strategy or combination of strategies is best depends on the compilation and linkage

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.

system used, the kind of application you are building, and the external constraints on the way you build systems. Generally, inline functions and other small functions that primarily call other template functions are candidates for inclusion into every compilation unit in which they are used. On an implementation with average support from the linker for template instantiation, doing this can speed up compilation and improve error messages.

Including a definition makes it vulnerable to having its meaning affected by macros and declarations in the context into which it is included. Consequently, larger template functions and template functions with nontrivial context dependencies are better compiled separately. Also, if the definition of a template requires a large number of declarations, these declarations can have undesirable side effects if they are included into the context in which the template is used.

I consider the approach of separately compiling template definitions and including declarations only in user code ideal. However, the application of ideals must be tempered by practical constraints, and separate compilation of templates is expensive on some implementations.

Whichever strategy is used, non-***inline static*** members (§C.13.1) must have a unique definition in some compilation unit. This implies that such members are best not used for templates that are otherwise included in many translation units.

One ideal is for code to work the same whether it is compiled as a single unit or separated into several separately translated units. That ideal should be approached by restricting a template definition's dependency on its environment rather than by trying to carry as much as possible of its definition context with it into the instantiation process.

13.8 Advice [temp.advice]

[1] Use templates to express algorithms that apply to many argument types; §13.3.
[2] Use templates to express containers; §13.2.
[3] Provide specializations for containers of pointers to minimize code size; §13.5.
[4] Always declare the general form of a template before specializations; §13.5.
[5] Declare a specialization before its use; §13.5.
[6] Minimize a template definition's dependence on its instantiation contexts; §13.2.5, §C.13.8.
[7] Define every specialization you declare; §13.5.
[8] Consider if a template needs specializations for C-style strings and arrays; §13.5.2.
[9] Parameterize with a policy object; §13.4.
[10] Use specialization and overloading to provide a single interface to implementations of the same concept for different types; §13.5.
[11] Provide a simple interface for simple cases and use overloading and default arguments to express less common cases; §13.5, §13.4.
[12] Debug concrete examples before generalizing to a template; §13.2.1.
[13] Remember to ***export*** template definitions that need to be accessible from other translation units; §13.7.
[14] Separately compile large templates and templates with nontrivial context dependencies; §13.7.
[15] Use templates to express conversions but define those conversions very carefully; §13.6.3.1.
[16] Where necessary, constrain template arguments using a ***constraint***`()` member function; §13.9[16].

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.

[17] Use explicit instantiation to minimize compile time and link time; §C.13.10.
[18] Prefer a template over derived classes when run-time efficiency is at a premium; §13.6.1.
[19] Prefer derived classes over a template if adding new variants without recompilation is important; §13.6.1.
[20] Prefer a template over derived classes when no common base can be defined; §13.6.1.
[21] Prefer a template over derived classes when built-in types and structures with compatibility constraints are important; §13.6.1.

13.9 Exercises [temp.exercises]

1. (∗2) Fix the errors in the definition of ***List*** from §13.2.5 and write out C++ code equivalent to what the compiler must generate for the definition of ***List*** and the function ***f*()**. Run a small test case using your hand-generated code and the code generated by the compiler from the template version. If possible on your system given your knowledge, compare the generated code.
2. (∗3) Write a singly-linked list class template that accepts elements of any type derived from a class ***Link*** that holds the information necessary to link elements. This is called an *intrusive list*. Using this list, write a singly-linked list that accepts elements of any type (a non-intrusive list). Compare the performance of the two list classes and discuss the tradeoffs between them.
3. (∗2.5) Write intrusive and non-intrusive doubly-linked lists. What operations should be provided in addition to the ones you found necessary to supply for a singly-linked list?
4. (∗2) Complete the ***String*** template from §13.2 based on the ***String*** class from §11.12.
5. (∗2) Define a ***sort*()** that takes its comparison criterion as a template argument. Define a class ***Record*** with two data members ***count*** and ***price***. Sort a ***vector<Record>*** on each data member.
6. (∗2) Implement a ***qsort*()** template.
7. (∗2) Write a program that reads **(*key*, *value*)** pairs and prints out the sum of the ***value***s corresponding to each distinct ***key***. Specify what is required for a type to be a ***key*** and a ***value***.
8. (∗2.5) Implement a simple ***Map*** class based on the ***Assoc*** class from §11.8. Make sure ***Map*** works correctly using both C-style strings and ***string***s as keys. Make sure ***Map*** works correctly for types with and without default constructors. Provide a way of iterating over the elements of a ***Map***.
9. (∗3) Compare the performance of the word count program from §11.8 against a program not using an associative array. Use the same style of I/O in both cases.
10. (∗3) Re-implement ***Map*** from §13.9[8] using a more suitable data structure (e.g., a red-black tree or a Splay tree).
11. (∗2.5) Use ***Map*** to implement a topological sort function. Topological sort is described in [Knuth,1968] vol. 1 (second edition), pg 262.
12. (∗1.5) Make the sum program from §13.9[7] work correctly for names containing spaces; for example, ‘‘thumb tack.’’
13. (∗2) Write ***readline*()** templates for different kinds of lines. For example (item,count,price).
14. (∗2) Use the technique outlined for ***Literate*** in §13.4 to sort strings in reverse lexicographical order. Make sure the technique works both for C++ implementations where ***char*** is ***signed*** and for C++ implementations where it is ***unsigned***. Use a variant of that technique to provide a sort that is not case-sensitive.

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.

15. (∗1.5) Construct an example that demonstrates at least three differences between a function template and a macro (not counting the differences in definition syntax).
16. (∗2) Devise a scheme that ensures that the compiler tests general constraints on the template arguments for every template for which an object is constructed. It is not sufficient just to test constraints of the form ''the argument ***T*** must be a class derived from ***My_base***.''

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.