Bjarne Stroustrup

THE C++ PROGRAMMING LANGUAGE

SECOND EDITION

The C++ Programming Language

Second Edition

Bjarne Stroustrup

AT&T Bell Laboratories
Murray Hill, New Jersey

ADDISON-WESLEY PUBLISHING COMPANY

Reading, Massachusetts · Menlo Park, California · New York
Don Mills, Ontario · Wokingham, England · Amsterdam · Bonn
Sydney · Singapore · Tokyo · Madrid · San Juan · Milan · Paris

Library of Congress Cataloging-in-Publication Data

Stroustrup, Bjarne.
The C++ programming language / Bjarne Stroustrup. -- 2nd ed.
p. cm.
Includes bibliographical references and index.
ISBN 0-201-53992-6
1. C++ (Computer program language) I. Title. II. Title: C plus plus programming language.
QA76.73.C15S79 1991
005. 13' 3--dc20 91-27307
CIP

Reprinted with corrections June, 1993

This book was typeset in Times and Courier by the author, using a Linotronic 200P phototypesetter and a DEC VAX 8550 running the 10th edition of the UNIX operating system.

11-MA-97 96 95 94

Preface

The road goes ever on and on.
– Bilbo Baggins

As promised in the first edition of this book, C++ has been evolving to meet the needs of its users. This evolution has been guided by the experience of users of widely varying backgrounds working in a great range of application areas. The C++ user-community has grown a hundredfold during the six years since the first edition of this book; many lessons have been learned, and many techniques have been discovered and/or validated by experience. Some of these experiences are reflected here.

The primary aim of the language extensions made in the last six years has been to enhance C++ as a language for data abstraction and object-oriented programming in general and to enhance it as a tool for writing high-quality libraries of user-defined types in particular. A "high-quality library," is a library that provides a concept to a user in the form of one or more classes that are convenient, safe, and efficient to use. In this context, *safe* means that a class provides a specific type-safe interface between the users of the library and its providers; *efficient* means that use of the class does not impose significant overheads in run-time or space on the user compared with hand-written C code.

This book presents the complete C++ language. Chapters 1 through 10 give a tutorial introduction; Chapters 11 through 13 provide a discussion of design and software development issues; and, finally, the complete C++ reference manual is included. Naturally, the features added and resolutions made since the original edition are integral parts of the presentation. They include refined overloading resolution, memory management facilities, and access control mechanisms, type-safe linkage, `const` and `static` member functions, abstract classes, multiple inheritance, templates, and exception handling.

C++ is a general-purpose programming language; its core application domain is

systems programming in the broadest sense. In addition, C++ is successfully used in many application areas that are not covered by this label. Implementations of C++ exist from some of the most modest microcomputers to the largest supercomputers and for almost all operating systems. Consequently, this book describes the C++ language itself without trying to explain a particular implementation, programming environment, or library.

This book presents many examples of classes that, though useful, should be classified as "toys." This style of exposition allows general principles and useful techniques to stand out more clearly than they would in a fully elaborated program, where they would be buried in details. Most of the useful classes presented here, such as linked lists, arrays, character strings, matrices, graphics classes, associative arrays, etc., are available in "bulletproof" and/or "goldplated" versions from a wide variety of commercial and non-commercial sources. Many of these "industrial strength" classes and libraries are actually direct and indirect descendants of the toy versions found here.

This edition provides a greater emphasis on tutorial aspects than did the first edition of this book. However, the presentation is still aimed squarely at experienced programmers and endeavors not to insult their intelligence or experience. The discussion of design issues has been greatly expanded to reflect the demand for information beyond the description of language features and their immediate use. Technical detail and precision have also been increased. The reference manual, in particular, represents many years of work in this direction. The intent has been to provide a book with a depth sufficient to make more than one reading rewarding to most programmers. In other words, this book presents the C++ language, its fundamental principles, and the key techniques needed to apply it. Enjoy!

Acknowledgments

In addition to the people mentioned in the acknowledgements section in the preface to the first edition, I would like to thank Al Aho, Steve Buroff, Jim Coplien, Ted Goldstein, Tony Hansen, Lorraine Juhl, Peter Juhl, Brian Kernighan, Andrew Koenig, Bill Leggett, Warren Montgomery, Mike Mowbray, Rob Murray, Jonathan Shopiro, Mike Vilot, and Peter Weinberger for commenting on draft chapters of this second edition. Many people influenced the development of C++ from 1985 to 1991. I can mention only a few: Andrew Koenig, Brian Kernighan, Doug McIlroy, and Jonathan Shopiro. Also thanks to the many participants of the "external reviews" of the reference manual drafts and to the people who suffered through the first year of X3J16.

Murray Hill, New Jersey *Bjarne Stroustrup*

Preface to the first Edition

Language shapes the way we think,
and determines what we can think about.
– B.L.Whorf

C++ is a general purpose programming language designed to make programming more enjoyable for the serious programmer. Except for minor details, C++ is a superset of the C programming language. In addition to the facilities provided by C, C++ provides flexible and efficient facilities for defining new types. A programmer can partition an application into manageable pieces by defining new types that closely match the concepts of the application. This technique for program construction is often called *data abstraction*. Objects of some user-defined types contain type information. Such objects can be used conveniently and safely in contexts in which their type cannot be determined at compile time. Programs using objects of such types are often called *object based*. When used well, these techniques result in shorter, easier to understand, and easier to maintain programs.

The key concept in C++ is *class*. A class is a user-defined type. Classes provide data hiding, guaranteed initialization of data, implicit type conversion for user-defined types, dynamic typing, user-controlled memory management, and mechanisms for overloading operators. C++ provides much better facilities for type checking and for expressing modularity than C does. It also contains improvements that are not directly related to classes, including symbolic constants, inline substitution of functions, default function arguments, overloaded function names, free store management operators, and a reference type. C++ retains C's ability to deal efficiently with the fundamental objects of the hardware (bits, bytes, words, addresses, etc.). This allows the user-defined types to be implemented with a pleasing degree of efficiency.

C++ and its standard libraries are designed for portability. The current implementation will run on most systems that support C. C libraries can be used from a C++

program, and most tools that support programming in C can be used with C++.

This book is primarily intended to help serious programmers learn the language and use it for nontrivial projects. It provides a complete description of C++, many complete examples, and many more program fragments.

Acknowledgments

C++ could never have matured without the constant use, suggestions, and constructive criticism of many friends and colleagues. In particular, Tom Cargill, Jim Coplien, Stu Feldman, Sandy Fraser, Steve Johnson, Brian Kernighan, Bart Locanthi, Doug McIlroy, Dennis Ritchie, Larry Rosler, Jerry Schwarz, and Jon Shopiro provided important ideas for development of the language. Dave Presotto wrote the current implementation of the stream I/O library.

In addition, hundreds of people contributed to the development of C++ and its compiler by sending me suggestions for improvements, descriptions of problems they had encountered, and compiler errors. I can mention only a few: Gary Bishop, Andrew Hume, Tom Karzes, Victor Milenkovic, Rob Murray, Leonie Rose, Brian Schmult, and Gary Walker.

Many people have also helped with the production of this book, in particular, Jon Bentley, Laura Eaves, Brian Kernighan, Ted Kowalski, Steve Mahaney, Jon Shopiro, and the participants in the C++ course held at Bell Labs, Columbus, Ohio, June 26-27, 1985.

Murray Hill, New Jersey *Bjarne Stroustrup*

Contents

8

Templates

Your quote here.
– B.Stroustrup

This chapter introduces the template concept that allows container classes, such as lists and associative arrays, to be simply defined and implemented without loss of static type checking or run-time efficiency. Similarly, templates allow generic functions, such as `sort()`, to be defined once for a family of types. A family of list classes is defined as an example of templates and their interaction with other language features. Some variants of a `sort()` function template is presented to demonstrate techniques for using templates to compose code from semi-independent parts. Finally, a simple associative array template is defined and used in a couple of small example programs.

8.1 Introduction

One of the most useful kinds of classes is the container class, that is, a class that holds objects of some (other) type. Lists, arrays, associative arrays, and sets are container classes. Specifying a container of objects of a single known type can be done with the facilities described in Chapters 5 and 7. For example §5.3.2 defined a set of `int`s. However, container classes have the interesting property that the type of objects they contain is of little interest to the definer of a container class, but of crucial importance to the user of a particular container. Thus we want to have the type of the contained object be an argument to a container class: The definer specifies the container class in terms of that argument, and the users specify what the type of contained objects is to be for each particular container (each object of the container class). The `Vector` template in §1.4.3 was an example of this.

This chapter first presents the notion of a template class by examining a simple `stack` template. Then a more complete and realistic example of a couple of related list templates is presented. Function templates and the rules for what can be a function template argument are stated. Finally, an associative array template is presented.

8.2 A Simple Template

A class template specifies how individual classes can be constructed much as a class declaration specifies how individual objects can be constructed. We can define a `stack` of elements of an arbitrary type:

```
template<class T>
class stack {
    T* v;
    T* p;
    int sz;

public:
    stack(int s) { v = p = new T[sz=s]; }
    ~stack() { delete[] v; }

    void push(T a) { *p++ = a; }
    T pop() { return *--p; }

    int size() const { return p-v; }
};
```

All run-time error checking has been left out for simplicity. Apart from that, the example is complete and realistic.

The `template <class T>` prefix specifies that a template is being declared and that an argument `T` of type *type* will be used in the declaration. After its introduction, `T` is used exactly like other type names. The scope of `T` extends to the end of the declaration that `template <class T>` prefixes. Note that `template<class T>` says that `T` is a *type* name; it need not actually be the name of a *class*. For `sc` below, `T` turns out to be `char`.

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:

```
stack<char> sc(100);      // stack of characters
```

defines an object `sc` of a class `stack<char>`.

Except for the special syntax of its name, `stack<char>` works exactly as if it had been defined:

```
class stack_char {
    char* v;
    char* p;
    int sz;
public:
    stack_char(int s) { v = p = new char[sz=s]; }
    ~stack_char() { delete[] v; }

    void push(char a) { *p++ = a; }
    char pop() { return *--p; }

    int size() const { return p-v; }
};
```

One can think of a template as a clever kind of macro that obeys the scope, naming, and type rules of C++. That would be an oversimplification, but it is an oversimplification that might help avoid some gross misunderstandings. In particular, use of a template need not imply any run-time mechanisms beyond what is used for an equivalent "hand-written" class, nor does it necessarily imply any savings in the amount of code generated.

It is usually a good idea to debug a particular class, such as `stack_char`, before turning it into a template such as `stack<T>`. Similarly, when trying to understand a template it is often useful to imagine its behavior for a particular type such as `int` or `shape*` before trying to comprehend the template in its full generality.

Given the class template declaration, `stacks` can now be defined and used like this:

```
stack<shape*> ssp(200);  // stack of pointers to shapes
stack<Point> sp(400);    // stack of Points

void f(stack<complex>& sc)     // 'reference to stack
                               // of complex' argument
{
    sc.push(complex(1,2));
    complex z = 2.5*sc.pop();

    stack<int>*p = 0;          // pointer to stack of ints
    p = new stack<int>(800); // stack of ints
                               // on the free store

    for (int i = 0; i<400; i++) {
        p->push(i);
        sp.push(Point(i,i+400));
    }

    // ...
}
```

Because the `stack` member functions were all inline, the only function calls

generated for this example were the ones generated for free store allocation and deallocation.

Template functions need not be inline; `stack` could equally well have been defined:

```
template<class T> class stack {
    T* v;
    T* p;
    int sz;
public:
    stack(int);
    ~stack();

    void push(T);
    T pop();

    int size() const;
};
```

In that case, definitions of the `stack` member function must be provided somewhere exactly as for non-template class member functions. Such functions are themselves parameterized by the type argument to their template class; thus these functions are defined by function templates. When defined outside the template class, this must be explicit. For example:

```
template<class T> void stack<T>::push(T a)
{
    *p++ = a;
}

template<class T> stack<T>::stack(int s)
{
    v = p = new T[sz=s];
}
```

Note that within the scope of `stack<T>` qualification with `<T>` is redundant so that `stack<T>::stack` is the name for the constructor.

It is the implementation's job – *not* the programmer's – to ensure that versions of template functions are generated for each argument type to the template. Thus for the example above, the implementation would generate definitions for the constructors for `stack<shape*>`, `stack<Point>`, and `stack<int>`, the destructors for `stack<shape*>` and `stack<Point>`, the `push()` functions for `stack<complex>`, `stack<int>`, `stack<Point>`, and the `pop()` function for `stack<complex>`. The generated functions will be perfectly ordinary member functions. For example:

```
void stack<complex>::push(complex a) { *p++ = a; }
```

differs from "ordinary member functions" only in the syntax for the class name.

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. When a function definition is needed for a class template member function for a particular type, it is the implementation's job to find the template for the member function and generate the appropriate version. An implementation may require the programmer to help find template source by following some convention.

It is important to write templates so that they have as few dependencies on global information as possible. The reason is that a template will be used to generate functions and classes based on unknown types and in unknown contexts. Almost any subtle context dependency will surface as a debugging problem to a programmer who is unlikely to be the original author of the template. The rule of avoiding references to global names as far as possible should be taken extra seriously in template design.

8.3 List Templates

When writing a realistic collection class, one often needs to deal with relationships between the classes involved in the implementation, with memory management issues, and with the need to iterate over a collection. It is also common to design several related classes together (§12.2). As an example, we will present a family of singly linked list classes and templates.

8.3.1 An Intrusive List

First, we will build a simple list that relies on a link field in objects put onto the list. Later, we use that list as a building block for a more general list that does not require a link field in objects put onto it. The class declarations with only their public functions will be presented first; the implementation will be presented in the next section. The idea is to avoid obscuring the design points with implementation details.

First we define a type `slink`, that is, a link in a singly linked list:

```
struct slink {
    slink* next;
    slink() { next=0; }
    slink(slink* p) { next = p; }
};
```

We can now define a class that can contain objects of any class derived from `slink`:

```
class slist_base {
    // ...
public:
    int insert(slink*);   // add at head of list
    int append(slink*);   // add at tail of list
    slink* get();         // remove and return head of list
    // ...
};
```

This class is intrusive because it can be used only if the elements provide the `slink` as a handle for `slist_base` to use. The name `slist_base` indicates that the class will be used as a base for singly linked list classes. As ever, when one designs a family of related entities there is a problem selecting names for the various members of the family. Since class names cannot be overloaded the way function names can, we cannot use overloading to reduce the name proliferation.

An `slist_base` can be used like this:

```
void f()
{
    slist_base slb;
    slb.insert(new slink);
    // ...
    slink* p = slb.get();
    // ...
    delete p;
}
```

However, because `slinks` don't carry information (beyond their identity), this is not very interesting. To use a `slist_base`, one needs to derive a useful class from `slink`. In a compiler one might have a `name` node that needs to be on a list:

```
class name : public slink {
    // ...
};

void f(const char* s)
{
    slist_base slb;
    slb.insert(new name(s));
    // ...
    name* p = (name*)slb.get();
    // ...
    delete p;
}
```

This works, but because `slist_base` is defined in terms of `slinks` and not names, it is necessary to use an explicit cast to convert the `slink*` returned by `slist_base::get()` into a `name*`. This is inelegant. In a large program with many lists and many classes derived from `slink`, it is also error prone. What we would like is a type-safe version of `slist_base`:

```
template<class T>
class Islist : private slist_base {
public:
    void insert(T* a) { slist_base::insert(a); }
    T* get() { return (T*) slist_base::get(); }
    // ...
};
```

The cast inside `Islist::get()` is perfectly reasonable and safe because class `Islist` ensures that every object on the list really is of type `T` or of a type derived from `T`. Note that `slist_base` is a `private` base class of `Islist`. We do not want users accidentally messing around with the unsafe implementation details.

The name `Islist` stands for "intrusive singly linked list." This template can be used like this:

```
void f(const char* s)
{
    Islist<name> ilst;
    ilst.insert(new name(s));
    // ...
    name* p = ilst.get();
    // ...
    delete p;
}
```

Attempted misuses are caught at compile time:

```
class expr : public slink {
    // ...
};

void g(expr* e)
{
    Islist<name> ilst;
    ilst.insert(e); // error: Islist<name>::insert()
                    // expects a name*
    // ...
}
```

There are several important things to note about the example so far. First, the scheme is type safe (barring silly mistakes in the very limited context of the access functions in `Islist`). Second, type safety is achieved without the expenditure of time or space because the access functions in `Islist` are trivial inline functions. Third, because all the real work is done by the – yet to be presented – implementation of `slist_base`, there is no replication of code, and the source code of the implementation (the `slist_base` functions) need not be available to a user. This is considered commercially important by some. It also provides a separation between an interface and its implementation so that re-implementation without requiring recompilation of user code becomes possible. Finally, a simple intrusive list is close to optimal in time and space. In other words, this strategy has near optimal properties of time, space, data hiding, and type checking while providing great flexibility and economy of expression.

However, an object can be on an `Islist` only provided it is derived from `slink`. This implies that we cannot have an `Islist` of `int`s, that we cannot have a list of some previously defined type that is not based on `slink`, and that having an object on two `Islists` takes some work (§6.5.1).

8.3.2 A Non-intrusive List

After our "digression" into the building and use of intrusive lists, we can proceed to building a non-intrusive list, that is, a list that does not require its elements to provide facilities to help the implementation of the list class. Because we no longer can assume that an object on the list has link field, the list implementation will have to provide one:

```
template<class T>
struct Tlink : public slink {
    T info;
    Tlink(const T& a) : info(a) { }
};
```

A `Tlink<T>` holds a copy of an object of type `T` in addition to the link field provided by its base class `slink`. Note that the use of the initializer `info(a)`, rather than the assignment `info=a`, is essential for efficient operation for types with non-trivial copy constructors and assignment operators (§7.11). For such a type – say `String` – defining the constructor,

```
Tlink(const T& a) { info = a; }
```

would have caused a default `String` to be constructed and then assigned to.

Given this link class and the `Islist` class, the definition of the non-intrusive list is almost trivial:

```
template<class T>
class Slist : private slist_base {
public:
    void insert(const T& a)
        { slist_base::insert(new Tlink<T>(a)); }
    void append(const T& a)
        { slist_base::append(new Tlink<T>(a)); }
    T get();
    // ...
};


template<class T>
T Slist<T>::get()
{
    Tlink<T>* lnk = (Tlink<T>*) slist_base::get();
    T i = lnk->info;
    delete lnk;
    return i;
}
```

The use of `Slist` is as simple as the use of `Islist`. The difference is that it is possible to have an object on an `Slist` without first deriving its class from `slink` and to have an object on two lists:

```
void f(int i)
{
    Slist<int> lst1;
    Slist<int> lst2;

    lst1.insert(i);
    lst2.insert(i);
    // ...

    int i1 = lst1.get();
    int i2 = lst1.get();
    // ...
}
```

However, an intrusive list, such as `Islist`, does have a consistent advantage in run-time efficiency and most often in compactness: Each time we put an object on an `Slist`, the list needs to allocate a `Tlink` object; each time we take an object off an `Slist`, the list needs to deallocate a `Tlink` object; and in each case a `T` is copied. Where the overhead is a problem, two things can be done. First, `Tlink` is a prime candidate for a near-optimal special-purpose allocator as described in §5.5.6; this will reduce the run-time overhead to something that is most often acceptable. Second, it is often a good idea to keep objects on a "primary list" that is intrusive and to use non-intrusive lists only where membership of several lists is needed:

```
void f(name* p)
{
    Islist<name> lst1;
    Slist<name*> lst2;

    lst1.insert(p);  // link through object '*p'
    lst2.insert(p);  // use separate link object to hold 'p'
    // ...
}
```

Naturally, such tricks can typically be played only within a particular component of a program (to avoid confusion about the types of lists used in inter-component interfaces) but *that* is exactly where run-time efficiency and compactness games are worth playing.

Because of the copying of the argument to `insert()` in the `Tlink` constructor, `Slist` is suitable only for small objects such as integers, `complex` numbers, and pointers. For objects where copying is expensive or unacceptable for semantic reasons, it is often a good idea to put pointers on the list rather than the objects themselves. This was done for `lst2` in `f()` above.

Note that because an argument to `Slist::insert()` is copied, passing an object of a derived class to an `insert()` function expecting an object of a base class will not work as (naively) expected:

```
class smiley : public circle { /* ... */ };

void g1(Slist<circle>& olist, const smiley& grin)
{
    olist.insert(grin);  // trap!
}
```

Only the `circle` part of the `smiley` will be stored. Note that this nasty problem is detected by the compiler in the case where it is most likely to occur. Had the base class in question been an abstract class, the compiler would have refused to "slice" the object of the derived class:

```
void g2(Slist<shape>& olist, const circle& c)
{
    olist.insert(c);  // error: attempt to create
                      // object of abstract class
}
```

Pointers must be used to avoid the problem of slicing:

```
void g3(Slist<shape*>& plist, const smiley& grin)
{
    plist.insert(&grin);  // fine
}
```

Don't use a reference as an argument to a class template. For example:

```
void g4(Slist<shape&>& rlist, const smiley& grin)
{
    rlist.insert(grin);  // error: generated code contains
                         // reference to reference (Shape&&)
}
```

References used like that most often cause type errors when the template is expanded. In this case the expansion of

```
Slist::insert(T&);
```

gives rise to the illegal declaration

```
Slist::insert(shape&&);
```

A reference is not an object, so it is not possible to have a reference to a reference.

Since lists of pointers are so useful it is a good idea to name them specifically:

```
template<class T>
class Splist : private Slist<void*> {
public:
    void insert(T* p) { Slist<void*>::insert(p); }
    void append(T* p) { Slist<void*>::append(p); }
    T* get() { return (T*) Slist<void*>::get(); }
};
```

```
template<class T>
class Isplist : private slist_base {
public:
    void insert(T* p) { slist_base::insert(p); }
    void append(T* p) { slist_base::append(p); }
    T* get() { return (T*) slist_base::get(); }
};
```

This also improves type checking and further reduces code replication.

It is often useful for the element type of a template itself to be a template class. For example, a sparse matrix of dates could be defined like this:

```
typedef Slist< Slist<date> > dates;
```

Please note the use of spaces here. Leaving out the space between the first and the second > would cause a syntax error when >> in

```
typedef Slist<Slist<date>> dates;
```

was interpreted as a right shift operator. As ever, a name introduced by a `typedef` is a synonym for the type it names and not a new type. Typedefs can be useful for longer template class names just as they are for other longish type names.

Note that a template argument used in several ways in a template should be mentioned once only in the list of template arguments. Thus, a template using a `T` object and a list of `T`'s is defined like this

```
template<class T> class mytemplate {
    T obj;
    Slist<T> slst;
    // ...
};
```

and *not* like this

```
template<class T, class Slist<T> > class mytemplate {
    T obj;
    Slist<T> slst;
    // ...
};
```

The rules for what can be a template argument can be found in §8.6 and §r.14.2.

8.3.3 A List Implementation

Implementing the `slist_base` functions is straightforward. The only real problem is what to do in case of an error, for example, what to do in case a user tries to `get()` something off an empty list. This will be handled by providing an error function, `slist_handler()`. Further strategies relying on exceptions will be discussed in Chapter 9.

The complete declaration of class `slist_base` is:

```
class slist_base {
    slink* last;  // last->next is head of list
public:
    void insert(slink* a);  // add at head of list
    void append(slink* a);  // add at tail of list
    slink* get();           // return and remove head

    void clear() { last = 0; }

    slist_base() { last = 0; }
    slist_base(slink* a) { last = a->next = a; }

    friend class slist_base_iter;
};
```

Storing a pointer to the last element of the circular list enables simple implementation of both an `append()` and an `insert()` operation:

```
void slist_base::insert(slink* a)  // add to head of list
{
    if (last)
        a->next = last->next;
    else
        last = a;
    last->next = a;
}
```

Note that `last->next` is the first element on the list.

```
void slist_base::append(slink* a)  //  add to tail of list
{
    if (last) {
        a->next = last->next;
        last = last->next = a;
    }
    else
        last = a->next = a;
}

slink* slist_base::get()  // return and remove head of list
{
    if (last == 0) slist_handler("get from empty slist");
    slink* f = last->next;
    if (f == last)
        last = 0;
    else
        last->next = f->next;
    return f;
}
```

For flexibility, it is a good idea to have `slist_handler` be a pointer to

function rather than a function. The call

```
slist_handler("get from empty list");
```

will then be equivalent to

```
(*slist_handler)("get from empty list");
```

As in the case of the `new_handler()` (§3.2.6) it is useful to provide a function

```
typedef void (*PFV)(const char*);

PFV set_slist_handler(PFV a)
{
    PFV old = slist_handler;
    slist_handler = a;
    return old;
}

PFV slist_handler = &default_slist_handler;
```

to help users manage their handlers. Exceptions, as described in Chapter 9, provide both an alternative way of handling errors and a way of implementing an `slist_handler`.

8.3.4 Iteration

Class `slist_base` provides no facilities for looking into a list, only the means for inserting and deleting members. It does, however, declare a class `slist_base_iter` to be a friend, so we can declare a suitable iterator. Here is one in the style presented in §7.8:

```
class slist_base_iter {
    slink* ce;       // current element
    slist_base* cs;  // current list
public:
    inline slist_base_iter(slist_base& s);
    inline slink* operator()();
};


slist_base_iter::slist_base_iter(slist_base& s)
{
    cs = &s;
    ce = cs->last;
}
```

```
slink* slist_base_iter::operator()()
    // return 0 to indicate end of iteration
{
    slink* ret = ce ? (ce=ce->next) : 0;
    if (ce == cs->last) ce = 0;
    return ret;
}
```

From this, iterators for `Slist` and `Islist` are easily constructed. First we must declare the iterators `friends` of their respective collection classes:

```
template<class T> class Islist_iter;

template<class T> class Islist {
    friend class Islist_iter<T>;
    // ...
};

template<class T> class Slist_iter;

template<class T> class Slist {
    friend class Slist_iter<T>;
    // ...
};
```

Note the way the names of the iterators are introduced without defining their template classes. This is the way one handles mutual dependencies between templates.

Next we can define the iterators:

```
template<class T>
class Islist_iter : private slist_base_iter {
public:
    Islist_iter(Islist<T>& s) : slist_base_iter(s) { }
    T* operator()()
        { return (T*) slist_base_iter::operator()(); }
};

template<class T>
class Slist_iter : private slist_base_iter {
public:
    Slist_iter(Slist<T>& s) : slist_base_iter(s) { }
    inline T* operator()();
};

template<class T> T* Slist_iter<T>::operator()()
{
        Tlink<T>* lnk = (Tlink<T>*) slist_base_iter::operator()(
        return lnk ? &lnk->info: 0;
}
```

Note that we again used the trick of deriving a family of classes (that is, a class template) from a unique base class. This uses inheritance to express commonality and to prevent unnecessary code replication. The importance of avoiding code replication in the implementation of simple, frequently used classes such as lists and iterators cannot be overstated. Such iterators can be used like this:

```
void f(name* n)
{
    Islist<name> lst1;
    Slist<name> lst2;

    lst1.insert(n);
    lst2.insert(n);
    // ...

    Islist_iter<name> iter1(lst1);
    const name* p;
    while (p=iter1()) {
        Slist_iter<name> iter2(lst2);
        const name* q;
        while (q=iter2()) {
            if (p == q) cout << "found " << *p << '\n';
        }
    }
}
```

There are several techniques for providing iteration for container classes. A designer of a program or a library will have to choose one style and stick to it. The style presented above is sometimes deemed "too cute." A less cute variant can be had by simply renaming `operator()()` as `next()`. Both variants have the property that cooperation between the iterator class and its container class is assumed so that it is possible to handle the case where elements are added to or removed from the container while an iterator is active. This and several other techniques would not be feasible if iteration depended on user code holding pointers to elements in the container. Typically, a container or its iterator supports a notion of resetting an iteration to "the beginning" and the notion of a "current element."

If the notion of a current element is provided by the container itself rather than by the iterator, the iteration becomes intrusive to the container in the same way links stored in element were to the contained objects. That is, it becomes hard to have two simultaneous iterations for a container, but the time and space properties of iteration approach optimum. For example:

```
class slist_base {
    // ...
    slink* last;  // last->next is head of list
    slink* current; // current element
public:
```

```
    // ...
    slink* head() { return last?last->next:0; }
    slink* current() { return current; }
    void   set_current(slink* p) { current = p; }
    slink* first() { set_current(head()); return current; }
    slink* next();
    slink* prev();
};
```

In the same way as both intrusive and non-intrusive lists could be used for the same object for space and time efficiency, both intrusive and non-intrusive iteration can be used for the same container:

```
void f(Islist<name>& ilst)
// dumb search for duplicates:
{
    list_iter<name> slow(ilst); // use iterator
    name* p;
    while (p = slow()) {
        ilst.set_current(p); // rely on current element
        name* q;
        while (q = ilist.next())
            if (strcmp(p->string,q->string) == 0)
                cout << "duplicate " << p << '\n';
    }
}
```

For yet another style of iterator see §8.8.

8.4 Function Templates

The use of template classes implies template member functions. In addition, global function templates, that is function templates that are not members of a class, can be defined. A function template defines a family of functions in the same way a class template defines a family of classes. This idea will be explored through a series of examples of how one might provide a `sort()` function. Each variant of `sort()` in the subsections below illustrates a general technique.

As ever, the focus of the discussion is program organization rather than algorithm design so a trivial algorithm is used. These variants of the `sort()` template are presented to demonstrate language features and useful techniques. The variants are not ordered according to "how good" they are, and there is also a lot to be said for the traditional non-template version (passing a pointer to a comparison function) in many contexts.

8.4.1 A Simple Global Function Template

Consider first the simplest `sort()` template:

```
template<class T> void sort(Vector<T>&);

void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
    sort(vi);   // sort(Vector<int>& v);
    sort(vc);   // sort(Vector<String>& v);
    sort(vi2);  // sort(Vector<int>& v);
    sort(vs);   // sort(Vector<char*>& v);
}
```

For each call, the argument type determines the sort function to be used. The programmer must provide a definition for the function template, and the language implementation must ensure that the proper variants of the template are created and called. For example, a simple bubble sort template might look like this:

```
template<class T> void sort(Vector<T>& v)
/*
    Sort the elements into increasing order

    Algorithm: bubble sort
*/
{
   unsigned int n = v.size();

   for (int i=0; i<n-1; i++)
       for (int j=n-1; i<j; j--)
           if (v[j] < v[j-1]) { // swap v[j] and v[j-1]
               T temp = v[j];
               v[j] = v[j-1];
               v[j-1] = temp;
           }
}
```

Please compare this to the bubble sort function from §4.6.9. The significant difference is that in the version here all the information needed is passed in the single argument `v`. Because the type of the elements is known (from the argument type), the comparison operator can be used directly rather than being passed as a pointer to function, and there is no need to mess around with the `sizeof` operator. This seems more elegant and is also more efficient than the traditional version. There is a problem, though. Some types do not have a `<` operator, and others, such as `char*`, have a `<` that does not do what is intended by the template function definition above. In the former case, an attempt to generate a version of `sort()` for such a type will fail (as one would hope it would); in the latter, surprising code will be generated.

To sort a `Vector` of `char*`s, we can simply specify a suitable implementation of `sort(Vector<char*>&)`:

```
void sort(Vector<char*>& v)
{
   unsigned int n = v.size();

   for (int i=0; i<n-1; i++)
       for (int j=n-1; i<j; j--)
           if (strcmp(v[j],v[j-1])<0) {
               // swap v[j] and v[j-1]
               char* temp = v[j];
               v[j] = v[j-1];
               v[j-1] = temp;
           }
}
```

Because a user-specified "special" definition of `sort()` is provided for vectors of character pointers, that special definition will be used, and no version `sort()` needs to be generated from the template for arguments of type `Vector<char*>&`. The ability for the programmer to provide separate definitions of template functions for specially important or "odd" types provides a valuable degree of flexibility and can be an important tool for performance tuning.

8.4.2 Adding Operations by Derivation

In the example above, the comparison function was "hard-wired" into the `sort()` function. An alternative would be to require the `Vector` class template to provide it. However, that requirement makes sense only for element types that have meaningful concepts of comparison. An traditional solution to that dilemma is to define `sort()` only for vectors that do have < defined:

```
template<class T> void sort(SortableVector<T>& v)
{
   unsigned int n = v.size();

   for (int i=0; i<n-1; i++)
       for (int j=n-1; i<j; j--)
           if (v.lessthan(v[j],v[j-1])) {
               // swap v[j] and v[j-1]
               T temp = v[j];
               v[j] = v[j-1];
               v[j-1] = temp;
           }
}
```

Class `SortableVector` might be defined like this:

```
template<class T> class SortableVector
    : public Vector<T>, public Comparator<T> {
public:
    SortableVector(int s) : Vector<T>(s) { }
};
```

To make this work we need to define a general `Comparator` class template:

```
template<class T> class Comparator {
public:
    static lessthan(T& a, T& b)
        { return a<b; }
    // ...
};
```

To handle the problem that < has the wrong semantics (for our purpose) for type `char*`, we define a special version of the class:

```
class Comparator<char*> {
public:
    static lessthan(const char* a, const char* b)
        { return strcmp(a,b)<0; }
    // ...
};
```

This declaration of a special version of a class template for `char*` closely mirrors the use of a special version of a function template for `char*` above. To have effect such a specialized version of a template class must be seen before its use. Otherwise, the class generated from the template will be used. Since a class must have exactly one definition in a program, attempting to use both a specialized version of a template class and the version generated from the template is an error.

Since we already have a special version of `Comparator` for `char*` we don't need a special version of `SortableVector` for `char*`, so we can finally write:

```
void f(SortableVector<int>& vi,
       SortableVector<String>& vc,
       SortableVector<int>& vi2,
       SortableVector<char*>& vs)
{
    sort(vi);
    sort(vc);
    sort(vi2);
    sort(vs);
}
```

Having two kinds of `Vectors` can be a bit of a nuisance, but at least `SortableVector` is derived from `Vector` so that a function that does not care about sorting need not worry about `SortableVectors`; the implicit conversion of a reference to a derived class to a reference to its public base takes care of that. The reason `SortableVector` was derived from both `Vector` and `Comparator`

(rather than adding functions to a class derived from `Vector` only) was simply that we had `Comparator` lying around from a previous example. This illustrates a style of composition found in larger libraries. A class like `Comparator` is a likely candidate for a library class, where it could be used to express the requirements for comparisons in many contexts.

8.4.3 Passing Operations as Function Arguments

An alternative to passing the comparison function as part of the `Vector` type is to pass it as a second argument to the `sort()` function. This second argument is an object of a class that specifies how comparison is to be done:

```
template<class T> void sort(Vector<T>& v, Comparator<T>& cmp)
{
    unsigned int n = v.size();

    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (cmp.lessthan(v[j],v[j-1])) {
                // swap v[j] and v[j-1]
                T temp = v[j];
                v[j] = v[j-1];
                v[j-1] = temp;
            }
}
```

This variant is a generalization of the traditional technique of passing the comparison operator as a pointer to function. We can now rewrite the user code:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
    Comparator<int> ci;
    Comparator<char*> cs;
    Comparator<String> cc;

    sort(vi,ci);    // sort(Vector<int>&);
    sort(vc,cc);    // sort(Vector<String>&);
    sort(vi2,ci);   // sort(Vector<int>&);
    sort(vs,cs);    // sort(Vector<char*>&);
}
```

Note that including the `Comparator` as a template argument ensures that inlining can be used for the `lessthan` operator. This technique is particularly useful if the function template requires several functions rather than a single comparison function and especially if the behavior of these functions are controlled by some data in the object they are part of.

8.4.4 Passing Operations Implicitly

In the previous section, the `Comparator` objects are not really used in the computation; they are simply "dummy arguments" used to drive the type system. Such a "dummy argument" is a useful and general – if not completely elegant – technique. However, where – as in the example above – an object is used to pass in operations *only*, that is, where the object's value and address is not used at all in the called function, the operations can be passed implicitly instead. For example:

```
template<class T> void sort(Vector<T>& v)
{
   unsigned int n = v.size();

   for (int i=0; i<n-1; i++)
       for (int j=n-1; i<j; j--)
           if (Comparator<T>::lessthan(v[j],v[j-1])) {
               // swap v[j] and v[j-1]
               T temp = v[j];
               v[j] = v[j-1];
               v[j-1] = temp;
           }
}
```

This allows our example to revert to its original version:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
    sort(vi);    // sort(Vector<int>&);
    sort(vc);    // sort(Vector<String>&);
    sort(vi2);   // sort(Vector<int>&);
    sort(vs);    // sort(Vector<char*>&);
}
```

The key advantage of this version and the previous (two versions) compared to the original simple `sort()` template is that the code for the sorting algorithm is separated from the code doing element-type-specific operations such as `lessthan`. This separation of concerns increases in importance as programs grow and is of particular interest in the context of library design where the library designer cannot know the template argument types and the users cannot know (or do not want to know) the details of the algorithms. In particular, had the `sort()` routine been a more complicated, optimized, "industrial strength" algorithm from a library, a user would have been reluctant to write the special version for type `char*` as was done in §8.4.1. Providing the special `char*` version of the `Comparator` class is trivial, though, and will have a variety of uses.

8.4.5 Adding Operations as Class Template Arguments

In some cases, having the connection between the `sort()` function template and the `Comparator` class template implicit can be a problem. The implicit connection can easily be overlooked and can also be hard to understand. Also, since it is "wired into" the `sort()` function it is not possible to use that `sort()` function to sort `Vectors` of a single type using different comparison criteria (see exercise 3 in §8.9). By wrapping the `sort()` function in a class we can allow the `Comparator` to be specified directly:

```
template<class T, class Comp> class Sort {
public:
    static void sort(Vector<T>&);
};


template<class T, class Comp>
void Sort<T,Comp>::sort(Vector<T>& v)
{
    unsigned int n = v.size();

    for (int i=0; i<n-1; i++)
        for (int j=n-1; i<j; j--)
            if (Comp::lessthan(v[j],v[j-1])) {
                T temp = v[j];
                v[j] = v[j-1];
                v[j-1] = temp;
            }
}
```

We can now select the appropriate `sort()` by qualifying it with a `Sort` class with appropriate element and comperator types:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
    Sort< int,Comparator<int> >::sort(vi);
    Sort< String,Comparator<String> >::sort(vc);
    Sort< int,Comparator<int> >::sort(vi2);
    Sort< char*,Comparator<char*> >::sort(vs);
}
```

This last variant is a powerful model for composition of code from separate parts. The example can even be further simplified by using the comparator type as the only template argument:

```
template<class Comp> class Sort {
public:
    class Comp::T; // Comp must have a member type T
    static void sort(Vector<Comp::T>&);
};
```

The `sort()` function will sort any `Vector` that `Comp` can compare elements of:

```
void f(Vector<int>& vi,
       Vector<String>& vc,
       Vector<int>& vi2,
       Vector<char*>& vs)
{
    Sort< Comparator<int> >::sort(vi);
    Sort< Comparator<String> >::sort(vc);
    Sort< Comparator<int> >::sort(vi2);
    Sort< Comparator<char*> >::sort(vs);
}
```

This implies that the comperator must give a name to its element type

```
template<class T> class Comparator {
public:
    typedef T T; // define Comparator<T>::T
    static int lessthan (T& a, T& b) {
        return a < b;
    }
    // ...
};
```

so that `Sort<Comp>::sort()` can refer to the element type as `Comp::T`.

8.5 Template Function Overloading Resolution

No conversions are applied to arguments on template functions. Instead, new versions are generated whereever possible. For example:

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

void f(int i, double d, complex z)
{
    complex z1 = sqrt(i);  // sqrt(int)
    complex z2 = sqrt(d);  // sqrt(double)
    complex z3 = sqrt(z);  // sqrt(complex)
    // ...
}
```

This will generate a `sqrt` function from the template for each of the three argument types. If the user wants something different – say a call of `sqrt(double)` given an `int` argument – explicit type conversion must be used:

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

void f(int i, double d, complex z)
{
    complex z1 = sqrt(double(i));  // sqrt(double)
    complex z2 = sqrt(d);  // sqrt(double)
    complex z3 = sqrt(z);  // sqrt(complex)
    // ...
}
```

Here, only `sqrt(double)` and `sqrt(complex)` definitions will be generated from the template.

A template function may be overloaded either by other functions of its name or by other template functions of that same name. Overloading resolution for template functions and other functions of the same name is done in three steps†:

[1] Look for an exact match (§r.13.2) on functions; if found, call it.

[2] Look for a function template from which a function that can be called with an exact match can be generated; if found, call it.

[3] Try ordinary overloading resolution (§r.13.2) for the functions; if a function is found, call it. If no match is found the call is an error.

In each case, if there is more than one alternative in the first step that finds a match, the call is ambiguous and is an error. For example:

```
template<class T>
    T max(T a, T b) { return a>b?a:b; };

void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);  // max(int,int)
    char m2 = max(c,d); // max(char,char)
    int m3 = max(a,c);  // error: cannot generate
                        // max(int,char)
}
```

Because no conversions are applied before selecting a template function to generate and call (rule 2 above), the last call cannot be resolved to `max(a,int(c))`. The programmer can resolve this by explicitly declaring `max(int,int)`. This would bring rule 3 into action:

```
template<class T>
    T max(T a, T b) { return a>b?a:b; };

int max(int,int);
```

† These rules are very strict and likely to be relaxed to allow pointer and reference conversions and possibly also other standard conversions. In that case, ambiguity control would be applied as always.

```
void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);  // max(int,int)
    char m2 = max(c,d); // max(char,char)
    int m3 = max(a,c);  // max(int,int)
}
```

There is no need to provide a definition for `max(int,int)`; it will be generated from the template by default.

The `max()` template could have been written to accept the example as originally written:

```
template<class T1, class T2>
    T1 max(T1 a, T2 b) { return a>b?a:b; };

void f(int a, int b, char c, char d)
{
    int m1 = max(a,b);  // int max(int,int)
    char m2 = max(c,d); // char max(char,char)
    int m3 = max(a,c);  // int max(int,char)
}
```

However, the C and C++ type rules for built-in types and operators are such that definition and use of such two-argument templates often get tricky. For example, the choice of `T1` as the result type would be wrong, or at least surprising, for a call

```
max(c,i);   // char max(char,int)
```

The use of two (or more) template arguments for functions taking a variety of arithmetic types can also lead to the generation of a surprising number of different functions and thus to a surprising amount of generated code. Forcing type conversion by explicit function declarations is most often a more manageable alternative.

8.6 Template Arguments

A template argument need not be a type name; see §r.14.2. In addition to type arguments, variable names, function names, and constant expressions can be used. In particular, integers can be useful as template arguments:

```
template<class T, int sz> class buffer {
    T v[sz];
    // ...
};
```

```
void f()
{
    buffer<char,128> buf1;
    buffer<complex,20> buf2;
    // ...
}
```

Making `sz` an argument of the template `buffer` itself rather than of its objects implies that the size of a `buffer` is known at compile time so that a `buffer` can be allocated without use of free store. This makes a template such as `buffer` useful for implementing container classes, where the use of free store can be the prime factor determining their run-time efficiency. For example, implementing a `string` class so that short strings are allocated on the stack is often a win because in many applications almost all strings are very short. The `buffer` template is useful in the implementation of such types.

Each template argument of a function template must affect the type of the function by affecting at least one argument type of functions generated from the template. This ensures that functions can be selected and generated based on their arguments. For example:

```
template<class T> void f1(T);    // fine
template<class T> void f2(T*);   // fine
template<class T> T f3(int);     // error
template<int i> void f4(int[][i]);  // error
template<int i> void f5(int = i);   // error
template<class T, class C> void f6(T);  // error
template<class T> void f7(const T&, complex);  // fine
template<class T> void f8(Vector< List<T> >);  // fine
```

In each case, the error is caused by a type argument not being used in a way that affects the argument types of the function.

There are no such constraints on arguments to class templates. The reason is that the arguments to a class template must be specified whenever an object of a template class is specified. On the other hand, for class templates we must answer the question, "When are two types the same?" Two template class names refer to the same class if their template names are identical and their arguments have identical values (modulo typedefs, constant expression evaluation, etc.). For example, consider the `buffer` template again:

```
template<class T, int sz>
class buffer {
    T v[sz];
    // ...
};
```

```
void f()
{
    buffer<char,20> buf1;
    buffer<complex,20> buf2;
    buffer<char,20> buf3;
    buffer<char,100> buf4;

    buf1 = buf2;  // error: type mismatch
    buf1 = buf3;  // fine
    buf1 = buf4;  // error: type mismatch

    // ...
}
```

Where non-type arguments are used for class templates, it is possible to construct some ambiguous looking constructs. For example:

```
template<int i>
class X { /* ... */ };

void f(int a, int b)
{
    X < a > b >; // X<a> b followed by a syntax error
                 // or X< (a>b) >; ?
}
```

This example is a syntax error because the first matching > terminates the template argument. In the unlikely case that you would want to specify a class template argument by a greater than expression, use parentheses: `X< (a>b) >`.

8.7 Derivation and Templates

As demonstrated above, the combination of derivation (inheritance) with templates can be a powerful tool. A template expresses a commonality across the types used as template arguments, and a base class expresses commonality of representation and calling interface. A few simple mistakes should be avoided.

Two types generated from a common template are different, and have no inheritance relationship, unless their template arguments are identical. For example:

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

Vector<int> v1;
Vector<short> v2;
Vector<int> v3;
```

Here `v1` and `v3` are of the same type and `v2` is of a completely different type. The fact that there is an implicit conversion from `short` to `int` does not imply that there is an implicit conversions from `Vector<short>` to `Vector<int>`:

```
v2 = v3;  // error: type mismatch
```

This is, I suspect, what one would expect since no built-in conversions exists between `int[]` and `short[]`.

Similarly:

```
class circle : public shape { /* ... */ };

Vector<circle*> v4;
Vector<shape*> v5;
Vector<circle*> v6;
```

Here `v4` and `v6` are of the same type and `v5` is of a completely different type. The fact that there are implicit conversions from `circle` to `shape` and from `circle*` to `shape*` does not imply that there are implicit conversions from `Vector<circle*>` to `Vector<shape*>` or from `Vector<circle*>*` to `Vector<shape*>*`:

```
v5 = v6;  // error: type mismatch
```

The reason is that, in general, the structure (representation) of one class generated from a class template is such that an inheritance relationship cannot be assumed. For example, the generated class may contain an object of the argument type rather than just a pointer. Furthermore, had such conversions been allowed, we would have been vulnerable to a hole in the type system. For example:

```
void f(Vector<circle>* pc)
{
    Vector<shape>* ps = pc;  // error: type mismatch
    (*ps)[2] = new square;   // put a square peg
                             // into a round hole
}
```

As shown with `Islist`, `Tlink`, `Slist`, `Splist`, `Islist_iter`, `Slist_iter`, and `SortableVector`, templates provide a useful way of deriving families of classes. Without templates the derivation of similar classes can become tedious and thus error prone. Conversely, without derivation, use of templates would imply massive replication of class template member function code, massive replication of declarative information in class templates, and massive replication of functions using the templates.

8.7.1 Specifying Implementation through Template Arguments

Container classes often have to allocate storage. Occasionally, it is necessary – or simply convenient – to give users the opportunity to chose between different allocation strategies and to supply their own. This can be done in several ways. One way is to use a template to compose a new class out of the interface to the desired container and an allocator class using the placement technique described in §6.7.2:

```
template<class T, class A> class Controlled_container
    : public Container<T>, private A {

    // ...
    void some_function()
    {
        // ...
        T* p = new(A::operator new(sizeof(T))) T;
        // ...
    }
    // ...
};
```

Here it is necessary to use a template because we are designing a container class; derivation from `Container<T>` is needed to allow a `Controlled_container` to be used as a container; and the use of the template argument `A` is necessary to allow a variety of allocators to be used. For example:

```
class Shared : public Arena { /* ... */ };
class Fast_allocator { /* ...*/ };
class Persistent : public Arena { /* ... */ };


Controlled_container<Process_descriptor,Shared> ptbl;

Controlled_container<Node,Fast_allocator> tree;

Controlled_container<Personnel_record,Persistent> payroll;
```

This is a general strategy for providing non-trivial implementation information for a derived class. It has the advantage of being systematic and allowing inlining to be used. It does tend to lead to extraordinarily long names, though. As usual, `typedef` can be used to introduce synonyms for type names of undesirable length. For example:

```
typedef
Controlled_container<Personnel_record,Persistent> pp_record;

pp_record payroll;
```

Typically, one would use a template to define a class such as `pp_record` only if the "implementation information" added is significant enough to make hand coding it into a derived class unattractive. Examples are a general (and possibly standard in some library) `Comparator` class template (§8.4.2) and non-trivial (and possibly standard in some library) `Allocator` classes. Note that the derivation in such examples has a distinct "main line" (here the `Container`) that provides the user interface and "side lines" that provide implementation details.

8.8 An Associative Array

An associative array is probably the most useful general non-built-in type. An associative array, often called a *map* and sometimes a *dictionary*, keeps pairs of values. Given the one value, called the *key*, one can access the other, called the *value*. An associative array can be thought of as an array where the index need not be an integer:

```
template<class K, class V> class Map {
    // ...
public:
    V& operator[](const K&); // find the V
                             // corresponding to K
                             // and return
                             // a reference to it
    // ...
};
```

Thus, a key of type `K` names a value of type `V`. We assume that keys can be compared using equality, ==, and the less-than operator, <, so that we can keep the array sorted. We also assume that keys and values can be copied. Note that a `Map` differs from the `assoc` type presented in §7.7 by making no further assumptions.

```
#include <String.h>
#include <iostream.h>
#include "Map.h"


int main()
{
    Map<String,int> count;
    String word;

    while (cin >> word) count[word]++;

    for (Mapiter<String,int> p = count.first(); p; p++)
        cout << p.value() << '\t' << p.key() << '\n';

    return 0;
}
```

The `String` was used to avoid having to worry about memory management and overflows the way one would have to with a `char*`. An iterator, `MapIter`, was used to visit (and write) each value in order; `MapIter` provides iteration by imitating pointers. Given

```
It was new. It was singular. It was simple. It must succeed.
```

as input this program produced:

```
4       It
1       must
1       new.
1       simple.
1       singular.
1       succeed.
3       was
```

There are, of course, many ways of designing an associative array, and given a definition of `Map` and its associated iterator class, there are many ways of implementing them. Here, a trivial implementation is chosen. It uses a linear search that makes it unsuitable for large arrays. An "industrial strength" implementation would be designed with criteria, such as speed of lookup and compactness of representation, in mind; see Exercise 4 in §8.9. Here, I simply use a doubly linked list of `Links`:

```
template<class K, class V> class Map;
template<class K, class V> class Mapiter;

template<class K, class V> class Link {
    friend class Map<K,V>;
    friend class Mapiter<K,V>;
private:
    const K key;
    V value;

    Link* pre;
    Link* suc;

    Link(const K& k, const V& v) : key(k), value(v) { }
    ~Link() { delete suc; } // delete all links recursively
};
```

Each `Link` holds a `(key,value)` pair. Friendship is used to ensure that `Links` can be created, manipulated, and destroyed only by the appropriate `Map` and iterator classes. Note the forward declarations of the `Map` and `Mapiter` class templates.

The `Map` template itself looks like this:

```
template<class K, class V> class Map {
    friend class Mapiter<K,V>;
    Link<K,V>* head;
    Link<K,V>* current;
    V def_val;
    K def_key;
    int sz;

    static K kdef(); // default K value
    static V vdef(); // default V value
```

```
        void find(const K&);
        void init() { sz = 0; head = 0; current = 0; }

    public:

        Map() : def_key(kdef()), def_val(vdef())
            { init(); }
        Map(const K& k, const V& d) : def_key(k), def_val(d)
            { init(); }
        ~Map() { delete head; } // delete all links recursively

        Map(const Map&);
        Map& operator= (const Map&);

        V& operator[] (const K&);

        int size() const { return sz; }
        void clear() { delete head; init(); }
        void remove(const K& k);

            // iteration functions:

        Mapiter<K,V> element(const K& k)
        {
            (void) operator[](k);  // move current to k
            return Mapiter<K,V>(this,current);
        }
        Mapiter<K,V> first() { return Mapiter<K,V>(this,head); }
        Mapiter<K,V> last();
    };


    template<class K, class V> K Map<K,V>::kdef()
        { static K k; return k; }
    template<class K, class V> V Map<K,V>::vdef()
        { static V v; return v; }
```

The elements are stored in a sorted doubly linked list. For simplicity, no attempt is made to minimize lookup time (see Exercise 4 in §8.9). The critical operation is `operator[]()`:

```
template<class K, class V>
V& Map<K,V>::operator[](const K& k)
{
    if (head == 0) {
        current = head = new Link<K,V>(k,def_val);
        current->pre = current->suc = 0;
        return current->value;
    }
```

```
        Link<K,V>* p = head;
        for (;;) {
            if (p->key == k) {  // found
                current = p;
                return current->value;
            }

            if (k < p->key) {   // insert before p
                current = new Link<K,V>(k,def_val);
                current->pre = p->pre;
                current->suc = p;
                if (p == head)  // becomes new head
                    head = current;
                else
                    p->pre->suc = current;
                p->pre = current;
                return current->value;
            }

            Link<K,V>* s = p->suc;
            if (s == 0) {       // insert after p (at end)
                current = new Link<K,V>(k,def_val);
                current->pre = p;
                current->suc = 0;
                p->suc = current;
                return current->value;
            }
            p = s;
        }
    }
```

The subscript operator returns a reference to the value corresponding to the key given as an argument. If no corresponding value is found, a new element with a default value is returned. This allows the subscript operator to be used on the left hand side of assignments. The default values for keys and values is set by the `Map` constructors. The subscript operator also sets the value `current` used by the iterators.

The implementation of the remaining member functions is left as an exercise for the reader:

```
template<class K, class V> void Map<K,V>::remove(const K& k)
{
    // see exercise section 8.10 [2]
}

template<class K, class V> Map<K,V>::Map(const Map<K,V>& m)
{
    // copy the map and all its elements
}
```

```
template<class K, class V>
Map<K,V>& Map<K,V>::operator=(const Map<K,V>& m)
{
    // copy the map and all its elements
}
```

Now all we need is a notion of iteration. A `Map` has member functions `first()`, `last()`, and `element(const K&)` that return an iterator positioned at the first element, the last element, and the element with key indicated by an argument, respectively. This makes sense because we keep the elements ordered by their keys.

A `Map` iterator `Mapiter` looks like this:

```
template <class K, class V> class Mapiter {
    friend class Map<K,V>;

    Map<K,V>* m;
    Link<K,V>* p;

    Mapiter(Map<K,V>* mm, Link<K,V>* pp)
        { m = mm; p = pp; }
public:
    Mapiter() { m = 0; p = 0; }
    Mapiter(Map<K,V>& mm);

    operator void*() { return p; }

    const K& key();
    V& value();

    Mapiter& operator--();  // prefix
    void operator--(int);   // postfix
    Mapiter& operator++();  // prefix
    void operator++(int);   // postfix
};
```

Once positioned, `Mapiter`'s `key()` and `value()` functions refer to the key and value of the element referred to by the iterator, respectively.

```
template<class K, class V> const K& Mapiter<K,V>::key()
{
    if (p) return p->key; else return m->def_key;
}

template<class K, class V> V& Mapiter<K,V>::value()
{
    if (p) return p->value; else return m->def_val;
}
```

In analogy to pointers, operators ++ and -- are provided for moving forward and backward in the `Map`:

```
Mapiter<K,V>& Mapiter<K,V>::operator--() // prefix decrement
{
    if (p) p = p->pre;
    return *this;
}

void Mapiter<K,V>::operator--(int)  // postfix decrement
{
    if (p) p = p->pre;
}

Mapiter<K,V>& Mapiter<K,V>::operator++() // prefix increment
{
    if (p) p = p->suc;
    return *this;
}

void Mapiter<K,V>::operator++(int)  // postfix increment
{
    if (p) p = p->suc;
}
```

The postfix operations were made to return no value because the cost of creating and returning a new `Mapiter` on each iteration could be significant and because the usefulness of doing that would be slight.

A `Mapiter` can be initialized to refer to the head of a `Map`:

```
template<class K, class V> Mapiter<K,V>::Mapiter(Map<K,V>& mm)
{
    m = &mm; p = m->head;
}
```

The conversion operator `operator void*()` returns zero if the iterator does not refer to an element, and non-zero otherwise. This means that one can test an iterator `iter` like this:

```
void f(Mapiter<const char*,Shape*>& iter)
{
    // ...

    if (iter) {
        // refers to element of map
    }
    else {
        // doesn't refer to element of map
    }

    // ...
}
```

The same technique is used to control stream I/O operations §10.3.2.

If an iterator doesn't refer to an element of a map, its `key()` and `value()` functions return references to the default objects.

In case you have now forgotten the purpose of all this, here is another little program using a `Map`. We will assume that the input is a list of pairs of values such as:

```
hammer  2
nail    100
saw     3
saw     4
hammer  7
nail    1000
nail    250
```

We would like to sort this, to have values of matching items added, and to print them together with the sum of the values:

```
hammer  9
nail    1350
saw     7
----------------
total   1366
```

First we will write a function that reads lines and enters the items found into a table keyed on the first item on the line:

```
template<class K, class V>
    void readlines(Map<K,V>& key)
    {
        K word;
        while (cin >> word) {
            V val = 0;
            if (cin >> val)
                key[word] += val;
            else
                return;
        }
    }
```

Next we provide a simple `main()` program to call the `readlines()` function and print the resulting table:

```
main()
{
    Map<String,int> tbl("nil",0);
    readlines(tbl);

    int total = 0;
    for (Mapiter<String,int> p(tbl); p; ++p) {
        int val = p.value();
        total += val;
        cout << p.key() << '\t' << val << '\n';
    }

    cout << "----------------\n";
    cout << "total\t" << total << '\n';
}
```

8.9 Exercises

1. (*2) Write a set of doubly linked lists to complement the family of singly linked lists defined in §8.3.
2. (*3) Define a `String` template that takes the character type as a template argument and show how it could be used for both "ordinary characters" and a hypothetical `lchar` class (supposed to represent characters in a non-English or extended character set). Make sure that people who use this template does not suffer significantly in time, space, or convenience compared to an ordinary string class.
3. (*1.5) Define a class `Record` with two data members `count` and `price`. Sort a `Vector` or `Records` on each field. Do not modify the sort function or the `Vector` template.
4. (*2) Complete the `Map` class template by defining the missing member functions.
5. (*2) Re-implement `Map` from §8.8 using a doubly linked list class.
6. (*2.5) Re-implement `Map` from §8.8 using a balanced tree as described in Knuth vol. 3, §6.2.3.
7. (*2) Compare the performance of a `Map` where `Link` is implemented with and without a class specific allocator.
8. (*3) Compare the performance of the word count program from §8.8 against a program not using a `Map`. Use the same style of I/O in both cases. Do the comparison against several versions of the `Map` class including the one from your library (if yours provides a `Map`).
9. (*2.5) Use `Map` to implement a topological sort function. Topological sort is described in Knuth vol. 1 (second edition), pp 262.
10. (*2) Make the sum program from §8.8 work correctly for long names and names containing spaces such as "thumb tack."
11. (*2) Make readline templates for different kinds of lines. For example

(item,count,price).

12. (*2) Write a shell sort variant of the `Sort` class from §8.4.5 and show how to choose the sorting algorithm used through the template argument. Shell sort is described in Knuth vol. 3, §5.2.1.
13. (*1) Change `Map` and `Mapiter` so that postfix ++ and -- returns a `Mapiter`.
14. (*1.5) Use the templates-as-modules technique from §8.4.5 to write a sort function that will work on both `Vector<T>` and `T[]` inputs.