25

25 Templates

Many of the ideas and examples presented in the last few chapters have illustrated ways of making code "general purpose". The collection classes in Chapter 21 were general purpose storage structures for any kind of data item that could reasonably be handled using `void*` (pointer to anything) pointers. Classes like Queue and DynamicArray don't depend in any way on the data elements stored and so are completely general purpose and can be reused in any program. Class AVL and class `BTree` from Chapter 24 are a little more refined. However they are again fairly general purpose, they work with data items that are instances of any concrete class derived from their abstract classes `Keyed` and `KeyedStorable`.

Templates represent one more step toward making code truly general purpose. The idea of a template is that it defines operations for an unbounded variety of related data types.

Template functions

You can have "template functions". These define manipulations of some unspecified generic form of data element. The function will depend on the data elements being capable of being manipulated in particular ways. For example, a function might require that data elements support assignment (`operator=()`), equality and inequality tests (`operator==()`, `operator!=()`), comparisons (`operator<()`), and stream output (`operator<<(ostream&, ?&)` global operator function). But any data element, whether it be an `int` or an `Aircraft`, that supports all these operations is equally readily handled by the "function".

Template classes

Most often, template classes are classes whose instances look after other data elements. Collection classes are preeminent examples, their instances provide storage organization of other data elements. The member functions of these template classes are like individual template functions; they describe, in some general way, how to manipulate data elements. They work with any kind of data that supports the required operations.

25.1 A GENERAL FUNCTION AND ITS SPECIALIZATIONS

Example, an idea for a general function

To make things concrete, consider a simple example – the function `larger_of _two()`. This function takes two data elements as arguments, and returns the larger. In outline form, its code is as follows:

Outline code

```
Thing larger_of_two(Thing& thing1, Thing& thing2)
{
    if(thing1 > thing2)
            return thing1;
    else
            return thing2;
}
```

Versions for different data

You can imagine actual implementations for different types of data:

Instance of outline for short integer data

```
short larger_of_two(short& thing1, short& thing2)
{
    if(thing1 > thing2)
            return thing1;
    else
            return thing2;
}
```

or

Instance of outline for double data

```
double larger_of_two(double& thing1, double& thing2)
{
    if(thing1 > thing2)
            return thing1;
    else
            return thing2;
}
```

or even

Instance of outline for "Boxes"

```
Box larger_of_two(Box& thing1, Box& thing2)
{
    if(thing1 > thing2)
            return thing1;
    else
            return thing2;
}
```

which would be practical provided that class `Box` defines an `operator>()` function and permits assignment e.g.:

```
long Box::Size() { return fwidth*fheight*fbreadth; }
```

```
int Box::operator>(Box& other) { return Size() > other.Size(); }

…
Box b1;
Box b2;
…
Box bigBox = larger_of_two(b1,b2);
```

Assignment operator needed to use result of function

The italicised outline version of the function is a "template" for the other special purpose versions. The outline isn't code that can be compiled and used. The code is the specialized versions made up for each data type.

Here, the different specializations have been hand coded by following the outline and changing data types as necessary. That is just a tiresome coding exercise, something that the compiler can easily automate.

Of course, if you want the process automated, you had better be able to explain to the compiler that a particular piece of text is a "template" that is to be adapted as needed. This necessitates language extensions.

25.2 LANGUAGE EXTENSIONS FOR TEMPLATES

25.2.1 Template declaration

The obvious first requirement is that you be able to tell the compiler that you want a "function definition" to be an outline, a template for subsequent specialization. This is achieved using template specifications.

For the example `larger_of_two()` function, the C++ template would be:

```
template<class Thing>
Thing larger_of_two(Thing& thing1, Thing& thing2)
{
    if(thing1 > thing2)
            return thing1;
    else
            return thing2;
}
```

template keyword

The compiler reading the code first encounters the keyword `template`. It then knows that it is dealing with a declaration (or, as here, a definition) of either a template function or template class. The compiler suspends the normal processes that it uses to generate instruction sequences. Even if it is reading a definition, the compiler isn't to generate code, at least not yet. Instead, it is to build up some internal representation of the code pattern.

Template parameters

Following the `template` keyword, you get template parameters enclosed within a < begin bracket and a > end bracket. The outline code will be using some name(s) to represent data type(s); the example uses the name `Thing`. The compiler has to be warned to recognize these names. After all, it is going to have to adapt those parts of the code that involve data elements of these types.

You can have template functions and classes that are parameterized by more than one type of data. For example, you might want some "keyed" storage structure parameterized according to both the type of the data element stored and the type of the primary key used to compare data elements. (This might allow you to have data elements whose keys were really strings rather than the usual integers.) Here, multiple parameters are considered as an "advanced feature". They will not be covered; you will get to use such templates in your later studies.

Once it has read the `template <…>` header, the compiler will consume the following function or class declaration, or (member) function definition. No instructions get generated. But, the compiler remembers what it has read.

25.2.2 Template instantiation

The compiler uses its knowledge about a template function (or class) when it finds that function (class) being used in the actual code. For example, assuming that class `Box` and template function `larger_of_two()` have both been defined earlier in the file, you could have code like the following:

Need version for Box

Need version for double

```
int main()
{
    Box b1(6,4,7);
    Box b2(5,5,8);
    …
    Box bigbox = larger_of_two(b1,b2);
    cout << "The bigger box is "; bigbox.PrintOn(cout);
    …
    double d1, d2;
    cout << "Enter values : "; cin >> d1 >> d2;
    …
    cout << "The larger of values entered was : " <<
                larger_of_two(d1, d2) << endl;
    …
}
```

At the point where it finds bigbox = larger_of_two(b1,b2), the compiler notes that it has to generate the version of the function that works for boxes. Similarly, when it gets to the output statement later on, it notes that it must generate a version of the function that works for doubles.

When it reaches the end of the file, it "instantiates" the various versions of the template.

Figure 25.1 illustrates the basic idea (and is close to the actual mechanism for some compilers).

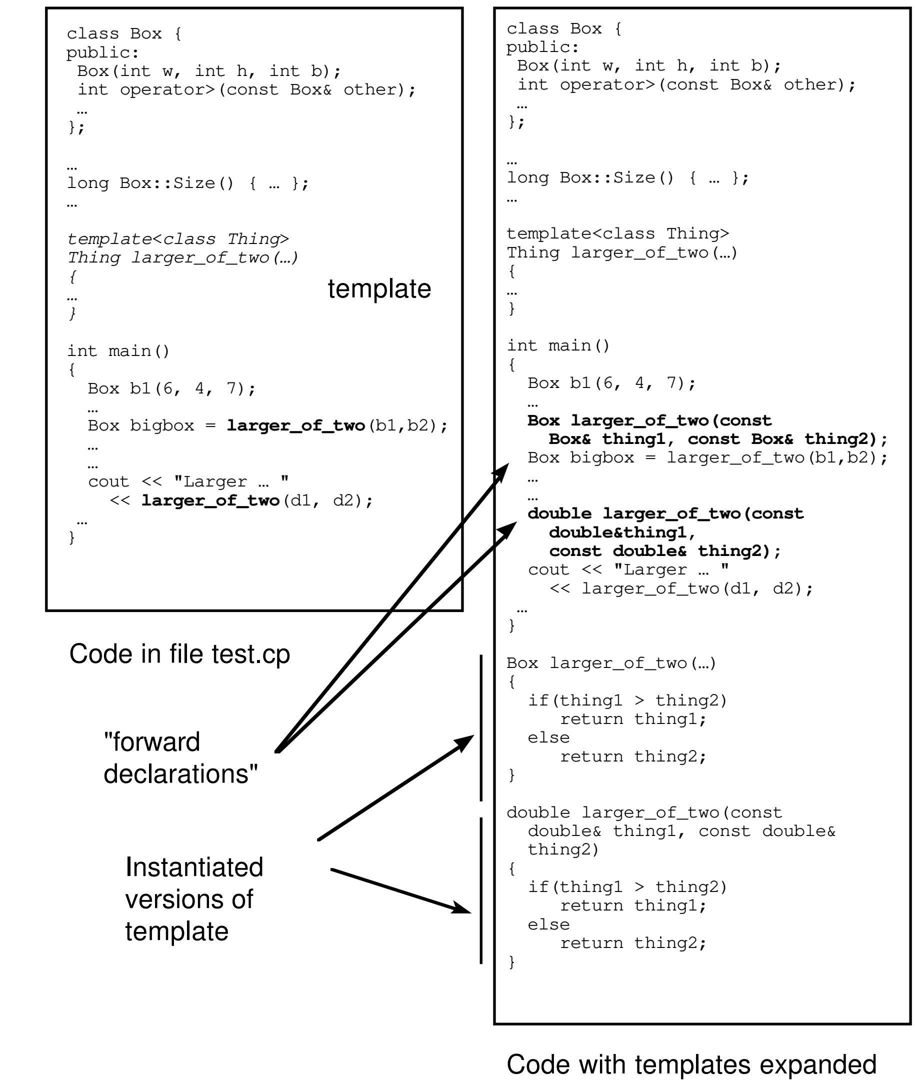

Figure 25.1 Illustration of conceptual mechanism for template instantiation.

The scheme shown in Figure 25.1 assumes that the compiler does a preliminary check through the code looking for use of templates. When it encounters uses of a template function, the compiler generates a declaration of a version specialized for the particular types of data used as arguments. When it gets to the end of the file, it uses the template to generate the specialized function, substituting the appropriate type (e.g. `double`) for the parametric type (`Thing`).

The specialized version(s) of the function can then be processed using the normal instruction generation mechanisms.

Watch out for compiler specific mechanisms

Templates are relatively new and the developers of compilers and linkers are still sorting out the best ways of dealing with them. The scheme shown in Figure 25.1 is fine if the entire program, with all its classes and template functions, is defined in a single file.

The situation is a little more complex when you have a program that is made up from many separately compiled and linked files. You may get references to template functions in several different files, but you probably don't want several different copies of "`double larger_of_two(const double& d1, const double& d2)`" – one copy for each file where this "function" is implicitly used.

Compiler "pragmas" to control instantiation

Compiler and linker systems have different ways of dealing with such problems. You may find that you have to use special compiler directives ("pragmas") to specify when and how you want template functions (and classes) to be instantiated. These directives are system's dependent. You will have to read the reference manuals for your IDE to see what mechanism you can use.

Another problem that you may find with your system is illustrated by the following:

```
template<class Whatsit>
void SortFun(Whatsit stuff[], int num_whatsists)
{
    // a standard sort algorithm
    …
}

int main()
{
    long array1[10], array2[12], array3[17];
    int n1, n2, n3;
    // Get number of elements and data for three arrays
    …
    // Sort array1
    SortFun(array1, n1);
    …
    // Deal with array2
    SortFun(array2, n2);
    …
}
```

Although in each case you would be sorting an array of long integers, the template instantiation mechanism might give you three versions of the code. One would be specialized for sorting an array of long integers with ten elements; the second would handle arrays with twelve elements; while the third would be specialized for arrays with seventeen elements.

This doesn't matter that much, it just wastes a little memory.

25.3 SPECIALIZED INSTANTIATIONS OF TEMPLATE CODE

Why does the compiler have to generate the specialized versions of template?

You should realize while that the code for the specialized data types may embody a similar pattern, the instruction sequences cannot be identical. All versions of the example `larger_of_two()` function take two address arguments (as previously explained references are handled internally as addresses). Thus, all versions will start with something like the following instruction sequence where the addresses of the two data elements are loaded into registers:

```
load a0 (address_register0) with stack_frame[…]
                                 Thing& thing1
load a1 (address_register1) with stack_frame[…]
                                 Thing& thing2
```

After that they have differences. For example, the generated code for the version with short integers will be something like:

```
load (integer register) r0 with two byte integer at a0
load (integer register) r1 with two byte integer at a1
compare r0, r1
branch if greater to pickthing1
// thing2 must be larger
store contents of r1 into 2-byte return field in stack frame
rts
pickthing1:
store contents of r0 into 2-byte return field in stack frame
rts
```

The code for the version specialized for doubles would be similar except that it would be working with 8-byte (or larger) data elements that get loaded into "floating point registers" and it would be a floating point comparison instruction that would be used.

There would be more significant differences in the case of `Boxes`. There the generated code would be something like:

```
push a0 onto stack
push a1 onto stack
call Box::operator>()
load r0 with contents of return-value from function
clean up stack
test r0
branch if greater to pickthing1
call routine to copy … bytes into return area of stackframe
rts
…
```

The idea maybe the same, but the realization, the instantiation differs.

25.4 A TEMPLATE VERSION OF QUICKSORT

The Quicksort function group from Chapter 13 provides a slightly more realistic example than `larger_of_two()`.

The Quicksort function illustrated earlier made use of two auxiliary functions. There was a function `Partition()` that shuffled data elements in a particular subarray and an auxiliary `SelectionSort()` function that was used when the subarrays were smaller than a given limit size.

The version in Chapter 13 specified integer data. But we can easily make it more general purpose by just recasting all the functions as templates whose arguments (and local temporary variables) are of some generic type `Data`:

```
const int kSMALL_ENOUGH = 15;
```

Template SelectionSort

```
template<class Data>
void SelectionSort(Data d[], int left, int right)
{
    for(int i = left; i < right; i++) {
            int min = i;
            for(int j=i+1; j<= right; j++)
                    if(d[j] < d[min]) min = j;
            Data temp = d[min];
            d[min] = d[i];
            d[i] = temp;
    }
}
```

Template Partition function

```
template<class Data>
int Partition(Data d[], int left, int right)
{
    Data val =d[left];
    int lm = left-1;
    int rm = right+1;
    for(;;) {
            do
                    rm--;
            while (d[rm] > val);

            do
                    lm++;
            while( d[lm] < val);

            if(lm<rm) {
                    Data tempr = d[rm];
                    d[rm] = d[lm];
                    d[lm] = tempr;
                    }
            else
```

```
                return rm;
            }
}

template<class Data>
void Quicksort(Data d[], int left, int right)
{
    if(left < (right-kSMALL_ENOUGH)) {
            int split_pt = Partition(d,left, right);
            Quicksort(d, left, split_pt);
            Quicksort(d, split_pt+1, right);
            }
    else SelectionSort(d, left, right);
}
```

Template QuickSort

The following test program instantiates two versions of the Quicksort group:

```
const int kBIG = 350;
int         data[kBIG];
double      data2[kBIG*2];

int main()
{
    int i;
    for(i=0;i <kBIG;i++)
            data[i] = rand() % 15000;

    Quicksort(data, 0, kBIG-1);

    for(i = 0; i < 100; i++) {
            cout << data[i] << ", ";
            if(5 == (i % 6)) cout << endl;
            }
    cout << "----" << endl;

    for(i=0;i <kBIG*2;i++)
            data2[i] = (rand() % 30000)/7.0;

    Quicksort(data2, 0, kBIG-1);
    for(i = 0; i < 100; i++) {
            cout << data2[i] << ", ";
            if(5 == (i % 6)) cout << endl;
            }

    cout << "----" << endl;

    …
```

Need Quicksort for integer

and a Quicksort for doubles

This test program runs successfully producing its two lots of sorted output:

```
54,        134,       188,       325,       342,       446,
```

```
524,       585,       609,       610,       629,       639,
…
3526,      3528,      3536,      3540,      ----

13.142,    25.285,    41.142,    49.571,    54.142,    58.142,
…
1129.714,  1134.142,  1135.857,  1140,      ----
```

Beware of silly errors

But you must be a little bit careful. How about the following program:

```
char *msgs[10] = {
    "Hello World",
    "Abracadabra",
    "2,4,6,8 who do we appreciate C++ C++ C++",
    "NO",
    "Zanzibar",
    "Zurich",
    "Mystery",
    "help!",
    "!pleh",
    "tenth"
};

int main()
{
    int i;
    for(i = 0; i < 10; i++)
            cout << msgs[i] << ", " << endl;
    cout << "----" << endl;

    Quicksort(msgs, 0,9);
    for(i = 0; i < 10; i++)
            cout << msgs[i] << ", " << endl;
    cout << "----" << endl;

    return 0;
}
```

The output this time is as follows:

```
Hello World,
Abracadabra,
2,4,6,8 who do we appreciate C++ C++ C++,
NO,
Zanzibar,
Zurich,
Mystery,
help!,
!pleh,
"tenth",
```

```
----
Hello World,
Abracadabra,
2,4,6,8 who do we appreciate C++ C++ C++,
NO,
Zanzibar,
Zurich,
Mystery,
help!,
!pleh,
"tenth",
----
```

A sort that didn't sort?

Problems! They aren't sorted; they are still in the original order! Has the sort routine got a bug in it? Maybe there is something wrong in the `SelectionSort()` part as its the only bit used with this small set of data.

What is really happening?

Another trial, one that might be more informative as to the problem:

Shuffle the data

```
int main()
{
    int i;

    char *msgs2[10];
    msgs2[0] = msgs[3]; msgs2[1] = msgs[2];
    msgs2[2] = msgs[8]; msgs2[3] = msgs[9];
    msgs2[4] = msgs[7]; msgs2[5] = msgs[0];
    msgs2[6] = msgs[1]; msgs2[7] = msgs[4];
    msgs2[8] = msgs[6]; msgs2[9] = msgs[5];
    for(i = 0; i < 10; i++)
            cout << msgs2[i] << ", " << endl;
    cout << "----" << endl;

    Quicksort(msgs2, 0,9);
    for(i = 0; i < 10; i++)
            cout << msgs2[i] << ", " << endl;

    return 0;
}
```

This produces the output:

```
NO,
2,4,6,8 who do we appreciate C++ C++ C++,
!pleh,
tenth,
help!,
Hello World,
Abracadabra,
Zanzibar,
Mystery,
```

```
Zurich,
----
Hello World,
Abracadabra,
2,4,6,8 who do we appreciate C++ C++ C++,
NO,
Zanzibar,
Zurich,
Mystery,
help!,
!pleh,
tenth,
```

Sorted back into the order of definition!

This time the call to `Quicksort()` did result in some actions. The data in the array `msgs2[]` have been rearranged. In fact, they are back in exactly the same order as they were in the definition of the array `msgs[]`.

What has happened is that the Quicksort group template has been instantiated to sort *an array of pointers to characters*. No great problems there. A pointer to character is internally the same as an unsigned long. You can certainly assign unsigned longs, and you can also compare them.

Sorting by location in memory!

We've been sorting the strings according to where they are stored in memory, and not according to their content! Don't get caught making silly errors like this.

You have to set up some extra support structure to get those strings sorted. The program would have to be something like the following code. Here a struct `Str` has been defined with associated `operator>()` and `operator<()` functions. A `Str` struct just owns a `char*` pointer; the comparison functions use the standard `strcmp()` function from the string library to compare the character strings that can be accessed via the pointers. A global `operator<<(ostream&, …)` function had also to be defined to allow convenient stream output.

```
struct Str {
    char* mptr;
    int operator<(const Str& other) const
           { return strcmp(mptr, other.mptr) < 0; }
    int operator>(const Str& other) const
           { return strcmp(mptr, other.mptr) > 0; }
};

ostream& operator<<(ostream& out, const Str& s) {
    out << s.mptr; return out;
}

Str msgs[10] = {
    "Hello World",
    …
    "tenth"
};
```

```
int main()
{
    int i;

    Quicksort(msgs, 0,9);
    for(i = 0; i < 10; i++)
            cout << msgs[i] << ", " << endl;

    return 0;
}
```

This program produces the output:

```
!pleh,
2,4,6,8 who do we appreciate C++ C++ C++,
Abracadabra,
Hello World,
Mystery,
NO,
Zanzibar,
Zurich,
help!,
tenth,
```

which is sorted. (In the standard ASCII collation sequence for characters, digits come before letters, upper case letter come before lower case letters.)

25.5 THE TEMPLATE CLASS "BOUNDED ARRAY"

Some languages, e.g. Pascal, have "bounded arrays". A programmer can define an array specifying both a lower bound and an upper bound for the index used to access the array. Subsequently, every access to the array is checked to make sure that the requested element exists. If an array has been declared as having an index range 9 to 17 inclusive, attempts to access elements 8 or 18 etc will result in the program being terminated with an "array bounds violation" error.

Such a structure is easy to model using a template `Array` class. We define a template that is parameterized by the type of data element that is to be stored in the array. Subsequently, we can have arrays of characters, arrays of integers, or arrays of any user defined structures.

An Array owns …

What does an `Array` own? It will have to remember the "bounds" that are supposed to exist; it needs these to check accesses. It will probably be convenient to store the number of elements, although this could information could always be recomputed An `Array` needs some space for the actual stored data. We will allocate this space on the heap, and so have a pointer data member in the `Array`. The space will be defined as an array of "things" (the type parameter for the class). Of course it is going to have to be accessed as a zero based array, so an `Array` will have to adjust the subscript.

An Array does …

An `Array` object will probably be asked to report its lower and upper subscript bounds and the number of elements that it owns; the class should provide access functions for these data. The main thing that an `Array` does is respond to the `[]` (array subscripting) operator.

The `[]` is just another operator. We can redefine `operator[]` to suit the needs of the class. We seem to have two ways of using elements of an array (e.g. `Counts` – an array of integers), as "right values":

```
n = Counts[6];
```

and as "left values":

```
Counts[2] += 10;
```

(The "left value" and "right value" terminology refers to the position of the reference to an array element relative to an assignment operator.) When used as a "right value" we want the value in the array element; when used as a "left value" we want the array element itself.

Reference to an array element

Both these uses are accommodated if we define **operator[]** as returning a reference to an array element. The compiler will sort out from context whether we need a copy of the value in that element or whether we are trying to change the element.

Assertions

How should we catch subscripting errors? The easiest way to reproduce the behaviour of a Pascal bounded array will be to have `assert()` macros in the accessing function that verify the subscript is within bounds. The program will terminate with an assertion error if an array is out of bounds.

Template class declaration

The template class declaration is as follows:

```
template<class DType>
class Array {
public:
    Array(int low_bnd, int high_bnd);
    ~Array();

    int     Size() const;
    int     Low() const;
    int     High() const;

    DType&  operator[](int ndx);
    void    PrintOn(ostream& os);
private:
```

```
        void operator=(const Array &a);
        Array(const Array &a);
        DType   *fData;
        int     flow;
        int     fhigh;
        int     fsize;
};
```

As in the case of a template function, the declaration starts with the keyword `template`, and then in `< >` brackets, we get the parameter specification. This template is parameterized by the type `DType`. This will represent the type of data stored in the array – `int`, `char`, `Aircraft`, or whatever.

Constructor

The class has a constructor that takes the lower and upper bounds, and a destructor. Actual classes based on this template will be "resource managers" – they create a structure in the heap to store their data elements. Consequently, a destructor is needed to get rid of this structure when an `Array` object is no longer required.

Access functions

There are three simple access functions, `Size()` etc, and an extra – `PrintOn()`. This may not really belong, but it is useful for debugging. Function `PrintOn()` will output the contents of the array; its implementation will rely on the stored data elements being streamable.

Array subscripting

The `operator[]()` function returns a "reference to a `DType`" (this will be a reference to integer, or reference to character, or reference to `Aircraft` depending on the type used to instantiate the template).

Copying of array disallowed

The declaration declares the copy constructor and `operator=()` function as private. This has been done to prevent copying and assignment of arrays. These functions will not be defined. They have been declared as `private` to get the compiler to disallow array assignment operations.

You could if you preferred make these functions public and provide definitions for them. The definitions would involve duplicating the data array of an existing object.

The data members are just the obvious integer fields needed to hold the bounds and a pointer to `DType` (really, a pointer to an array of `DType`).

Template class definition

All of the member functions must be defined as template functions so their definitions will start with the keyword `template` and the parameter(s).

The parameter must also be repeated with each use of the scope qualifier operator (`::`). We aren't simply defining the `Size()` or `operator[]()` functions of class `Array` because there could be several class `Arrays` (one for integers, one for `Aircraft` etc). We are defining the `Size()` or `operator[]()` functions of the class `Array` that has been parameterized for a particular data type. Consequently, the parameter must modify the class name.

The constructor is straightforward. It initializes the array bounds (first making certain that the bounds have been specified from low to high) and creates the array on the heap:

Constructor

```
template<class DType>
Array<DType>::Array(int low_bnd, int high_bnd)
{
    assert(low_bnd < high_bnd);
    fsize = high_bnd - low_bnd;
    fsize += 1; // low to high INCLUSIVE
    flow = low_bnd;
    fhigh = high_bnd;

    fData = new DType[fsize];
}
```

Destructor and simple access functions

The destructor simply gets rid of the storage array. The simple access functions are all similar, only `Size()` is shown:

```
template<class DType>
Array<DType>::~Array()
{
    delete [] fData;
}

template<class DType>
int Array<DType>::Size() const
{
    return fsize;
}
```

Output function

Function `PrintOn()` simply runs through the array outputting the values. The statement `os << fData[i]` has implications with respect to the instantiation of the array. Suppose you had class `Point` defined and wanted a bounded array of instances of class `Point`. When the compiler does the work of instantiating the `Array` for `Points`, it will note the need for a function `operator<<(ostream&, const Point&)`. If this function is not defined, the compiler will refuse to instantiate the template. The error message that you get may not be particularly clear. If you get errors when instantiating a template class, start by checking that the data type used as a parameter does support all required operations.

```
template<class DType>
void Array<DType>::PrintOn(ostream& os)
{
    for(int i=0;i<fsize;i++) {
            int j = i + flow;
            os << "[" << setw(4) << j << "]\t:";
            os << fData[i];
```

```
            os << endl;
            }
}
```

The `operator[]()` function sounds terrible but is actually quite simple:

Array indexing

```
template<class DType>
DType& Array<DType>::operator[](int ndx)
{
    assert(ndx >= flow);
    assert(ndx <= fhigh);

    int j = ndx - flow;

    return fData[j];
}
```

The function starts by using assertions to check the index given against the bounds. If the index is in range, it is remapped from `[flow … fhigh]` to a zero based array and then used to index the `fData` storage structure. The function simply returns `fData[j]`; the compiler knows that the return type is a reference and it sorts out from left/right context whether it should really be using an address (for left value) or contents (for right value).

Template class use

Code fragments using Array template

The following code fragments illustrate simple applications using the template `Array` class. The first specifies that it wants an `Array` that holds integers. Note the form of the declaration for `Array a`; it is an `Array`, parameterized by the type `int`.

```
#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>
#include <assert.h>

template<class DType>
class Array {
public:
…
};

// Definition of member functions as shown above
…

int main()
{
```

Define an Array of integers

```
    Array<int>    a(3,7);

    cout << a.Size() << endl;

    a[3] = 17; a[4] = 21; a[5] = a[3] + a[4];
    a[6] = -1; a[7] = 123;

    a.PrintOn(cout);

    return 0;
}
```

The second example uses an `Array<int>` for counters that hold letter frequencies:

```
int main()
{
    Array<int> letter_counts('a', 'z');
    for(char ch = 'a'; ch <= 'z'; ch++)
            letter_counts[ch] = 0;
    cout << "Enter some text,terminate with '.'" << endl;

    cin.get(ch);
    while(ch != '.') {
            if(isalpha(ch)) {
                    ch = tolower(ch);
                    letter_counts[ch] ++;
                    }
            cin.get(ch);
            }

    cout << endl << "-----" << endl;
    cout << "The letter frequencies are " << endl;
    letter_counts.PrintOn(cout);
    return 0;
}
```

(The character constants 'a' and 'z' are perfectly respectable integers and can be used as the low, high arguments needed when constructing the array.)

These two examples have both assumed that the template class is declared, and its member functions are defined, in the same file as the template is employed. This simplifies things, there are no problems about where the template should be instantiated. As in the case of template functions, things get a bit more complex if your program uses multiple files. The mechanisms used by your IDE for instantiating and linking with template classes will be explained in its reference manual..

25.6 THE TEMPLATE CLASS QUEUE

The collection classes from Chapter 21 are all candidates for reworking as template classes. The following is an illustrative reworking of class `Queue`. The `Queue` in Chapter 21 was a "circular buffer" employing a fixed sized array to hold the queued items.

The version in Chapter 21 held pointers to separately allocated data structures. So, its `Append()` function took a `void*` pointer argument, its `fData` array was an array of pointers, and function `First()` returned a `void*`.

Choice of data copies or data pointers

The new version is parameterized with respect to the data type stored. The declaration specifies a `Queue` of `Obj`s, and an `Obj` is anything you want it to be. We can still have a queue that holds pointers. For example, if we already have separately allocated `Aircraft` objects existing in the heap we can ask for a queue that makes `Obj` an `Aircraft*` pointer. But we can have queues that hold copies of the actual data rather than pointers to separate data. When working with something like `Aircraft` objects, it is probably best to have a queue that uses pointers; but if the data are of a simpler data type, it may be preferable to make copies of the data elements and store these copies in the `Queue`'s array. Thus, it is possible to have a queue of characters (which wouldn't have been very practical in the other scheme because of the substantial overhead of creating single characters as individual structures in the heap).

Type safety

Another advantage of this template version as compared with the original is that the compiler can do more type checking. Because the original version of `Queue::Append()` just wanted a `void*` argument, any address was good enough. This is a potential source of error. The compiler won't complain if you mix up all addresses of different things (`char*`, `Aircraft*`, addresses of functions, addresses of automatic variables), they are all just as good when all that is needed is a `void*`.

When something was removed from that queue, it was returned as a `void*` pointer that then had to be typecast to a useable type. The code simply had to assume that the cast was appropriate. But if different types of objects were put into the queue, all sorts of problems would arise when they were removed and used.

This version has compile time checking. The `Append()` function wants an `Obj` – that is an `Obj` that matches the type of the instantiated queue. If you ask for a queue of `Aircraft*` pointers, the only thing that you can append is an `Aircraft*`; when you take something from that queue it will be an `Aircraft*` and the compiler can check that you use it as such. Similarly, a queue defined as a queue of `char` will only allow you to append a `char` value and will give you back a `char`. Such static type checking can eliminate many errors that result from careless coding.

The actual declaration of the template is as follows:

```
template<class Obj>
class Queue {
public:
    Queue();

    void   Append(Obj newobj);
    Obj    First();
```

```
    int     Length() const;
    int     Full() const;
    int     Empty() const;

private:
    Obj     fdata[QSIZE];
    int     fp;
    int     fg;
    int     fcount;
};
```

The definitions of the member functions are:

Access functions

```
template<class Obj>
inline int Queue<Obj>::Length() const { return fcount; }

template<class Obj>
inline int Queue<Obj>::Full() const {
    return fcount == QSIZE;
}

template<class Obj>
inline int Queue<Obj>::Empty() const {
    return fcount == 0;
}
```

Constructor

```
template<class Obj>
Queue<Obj>::Queue()
{
    fp = fg = fcount = 0;
}
```

The `Append()` and `First()` functions use `assert()` macros to verify correct usage:

```
template<class Obj>
void Queue<Obj>::Append(Obj newobj)
{
    assert(!Full());
    fdata[fp] = newobj;
    fp++;
    if(fp == QSIZE)
            fp = 0;
    fcount++;
    return;
}

template<class Obj>
Obj Queue<Obj>::First(void)
{
    assert(!Empty());
```

```
        Obj temp = fdata[fg];
                fg++;
        if(fg == QSIZE)
                fg = 0;
        fcount--;
        return temp;
}
```

The following is an example code fragment that will cause the compiler to generate an instance of the template specialized for storing `char` values:

```
int main()
{
        Queue<char> aQ;
        for(int i=0; i < 100; i++) {
                int r = rand() & 1;
                if(r) {
                        if(!aQ.Full()) {
                                char ch = 'a' + (rand() % 26);
                                aQ.Append(ch);
                                cout << char(ch - 'a' + 'A');
                                }
                        }
                else {
                        if(!aQ.Empty()) cout << aQ.First();
                        }
                }
        return 0;
}
```