9 Template Programming

9.1 Designing a template class

In this section, we discuss the development of a template class with features similar to an STL container. (Koenig and Moo 2000, Chapter 11) provides a very similar presentation. For a comprehensive reference on templates, (Vandevoorde, Josuttis, and Gregor 2017) is recommended.

Our objective is to implement a class `Vec` that behaves in more or less the same way as the STL class `vector`. We will provide memory management, but in a somewhat less sophisticated way than Koenig and Moo's example. We will obtain the class declaration by filling in the blanks of this skeleton: 314

```
template <typename T>
class Vec
{
public:
    // interface
private:
    // hidden data
};
```

The data in the vector will be stored in a dynamic array of the template type, `T`. We need a pointer to the first element of the array and either: (a) the number of elements in the array, or (b) a pointer to one-past-the-last element of the array. Following STL conventions, we will adopt choice (b) and, of course, the user will see our pointers as iterators.

It would be possible to store exactly the number of elements that we need. This can be inefficient, however, if the user inserts elements into the array one at a time. Consequently, we provide a pointer to one-past-the-last element that is actually in use and another pointer to one-past-the-last element that is available for use. The three pointers are 315

1. `data` points to the first element
2. `avail` points to one-past-the-last element in use
3. `limit` points to one-past-the-last element of allocated memory

These three pointers define a ***class invariant*** for class `Vec`:

- `data` points to the first element
- `data` $\le$ `avail` $\le$ `limit`

- The semi-closed interval [`data`, `avail`) contains the allocated data

316 - The semi-closed interval [`avail`, `limit`) contains the allocated but uninitialized space

Figure 96 shows these pointers for an array in which 8 elements are in use and 13 elements are available altogether.

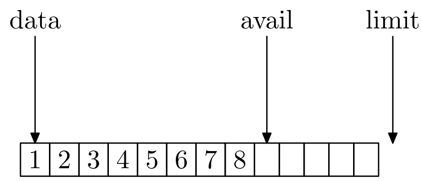

Figure 96: Pointers for class Vec

Following STL conventions, class `Vec` defines several types for its clients. These types hide, for example, the fact that `Vec` uses pointers to implement iterators. (We do not hide this fact because we are ashamed of using pointers, but rather to present a consistent abstraction to our clients.) The types we define are:

- `iterator` for the type of iterators
- `const_iterator` for the type of constant iterators
- `size_type` for the type of the size of a `Vec`
- `value_type` for the type of an element of a `Vec`

For `size_type`, we use `size_t` from namespace `std`.

We will provide three constructors: a default constructor; a constructor that specifies a size (number of elements) and an initial value for each element; and a copy constructor. The default constructor creates an empty vector. We declare the second constructor to be `explicit` to avoid accidental conversion. In the second constructor, the value of the element defaults to `T()`, which implies that the type `T` provided by the user must have a default constructor. Clearly, we will
317 also need a destructor. Figure 97 on the facing page shows the class declaration with the features
318 we have incorporated so far.

319 Figure 98 on page 180 shows implementations for the constructors and destructor. Because this
320 is a template class, these declarations are in the header file `vec.h` after the class declaration. The second constructor is inefficient: for example, if the caller assumes the default value for the second parameter, the constructor `T()` will be called $2n$ times: n times for `new T[n]` and another n times in the `for` statement. (Koenig and Moo 2000) show how to avoid this inefficiency by allocating uninitialized memory, but we will not bother with this improvement.

The copy constructor requires a function that returns the size (number of elements) of the `Vec`
321 object. Since this is also a useful function in general, we make it `public`:

```
size_type size() { return avail - data; }
```

```
template <typename T>
class Vec
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;

    Vec();
    explicit Vec(size_t n, const T & val = T());
    Vec(const Vec & other);
    ~Vec();

    // rest of interface

private:
    iterator data;
    iterator avail;
    iterator limit;
};
```

Figure 97: Declaration for class `Vec`: version 1

Since the assignment operator is somewhat similar to the copy constructor, we consider it next. Although they are similar, the difference between assignment and initialization is significant and must not be overlooked. Before assigning to a variable, we must destroy its current value. Also, as we have seen previously, we must provide the correct behaviour for the self-assignment, `x = x`. The implementation of `operator=` shown in Figure 99 on the following page takes care of all this. 322
The constructors and assignment operator introduce duplicated code into the implementation, but we can clean that up later.

9.1.1 Iterators

It is straightforward to define the functions `begin` and `end` that provide clients with iterators pointing to the first and one-past-the-last elements. Two versions are required, one returning an `iterator` and the other returning a `const_iterator`. 323

```
iterator begin() { return data; }
const_iterator begin() const { return data; }

iterator end() { return avail; }
const_iterator end() const { return avail; }
```

Since the operators `!=`, `++`, and `*` are all provided for pointers, these functions provide all that is needed for code such as this loop:

```
template <typename T>
Vec<T>::Vec<T>() : data(0), avail(0), limit(0) {}

template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
    : data(new T[n]), avail(data + n), limit(data + n)
{
    for (iterator p = data; p != avail; ++p)
        *p = val;
}

template <typename T>
Vec<T>::Vec<T>(const Vec<T> & other) :
    data(new T[other.size()]),
    avail(data + other.size()),
    limit(avail)
{
    const_iterator q = other.begin();
    for (iterator p = data; p != avail; ++p, ++q)
        *p = *q;
}

template <typename T>
Vec<T>::~Vec<T>()
{
    delete [] data;
}
```

Figure 98: Constructors and destructor for class `Vec`

```
template <typename T>
Vec<T> & Vec<T>::operator=(const Vec<T> & rhs)
{
    if (&rhs != this)
    {
        delete [] data;
        data = new T[rhs.size()];
        avail = data + rhs.size();
        limit = avail;
        const_iterator q = rhs.begin();
        for (iterator p = data; p != avail; ++p, ++q)
            *p = *q;
    }
    return *this;
}
```

Figure 99: Assignment operator for class `Vec`

```
for (Vec<int>::const_iterator it = v.begin(); it != v.end(); ++it)
    .... *it ....
```

We enable the user to subscript Vecs by implementing operator[]. Two overloads of this operator are required: 324

```
T & operator[](size_type i) { return data[i]; }
const T & operator[](size_type i) const { return data[i]; }
```

op[] vs. const op[]

The need for two versions arises as follows. We want a version of operator[] that allows us to use subscripted elements on the left of an assignment, as in a[i] = e. The first version of operator[] does this by returning a reference. But the compiler will not accept this function if it is applied to a const Vec, because the reference makes it possible to change the value of the object. Consequently, we need a second version that returns a const T & and promises not to change the object. Consider the following code:

```
Vec<int> v(5);
v[3] = 5;                    // (1)

const Vec<int> w(v);
n = w[2];                    // (2)
```

The statement labelled (1) fails if the first (non-const) version of operator[] is omitted, because it changes the value of v. The statement labelled (2) fails if the second (const) version of operator[] is omitted, because the compiler needs assurance that the call w[2] does not change the value of w.

Normally, we cannot provide two versions of a function with the same parameter list that differ only in their return type (see Section 7.4 on page 143). In this case, the object (*this) is an implicit first parameter, and the overloads distinguish a const Vec and a non-const Vec.[37]

As with similar STL functions, neither version of operator[] checks to see if the subscript is in range. Such a check could easily be added, perhaps as an assertion.

9.1.2 Expanding Vectors

The next function that we provide is push_back, which appends a new value to the end of the array. There are two cases: if there is space already allocated, we store the new element at the position indicated by avail and then increment avail. In the other case, avail = limit, and we must allocate more space. We will introduce a private function, grow, to find more space. Figure 100 shows the definitions of both functions. 325

326 The algorithm that we use for increasing the size of a Vec has important implications for efficiency. It is usually not possible simply to increase the size of a dynamic array, because the adjacent memory may already be allocated. Consequently, if we have an area of M bytes and we want to

[37]This problem can also be solved using the const_cast seen in Section 8.4: Simply provide the implementation for const, and re-use it by removing the constness using const_cast. This can avoid code duplication when operator[] is more complex. For details, see (Meyers 2005, p.23ff).

```
template<typename T>
void Vec<T>::push_back(const T & val)
{
    if (avail == limit)
        grow();
    *avail = val;
    ++avail;
}

template<typename T>
void Vec<T>::grow()
{
    size_type oldSize = avail - data;
    size_type newSpace = (oldSize == 0) ? 1 : 2 * (limit - data);
    iterator newData = new T[newSpace];
    iterator p = newData;
    for (const_iterator q = data; q != avail; ++q, ++p)
        *p = *q;
    delete [] data;
    data = newData;
    avail = data + oldSize;
    limit = data + newSpace;
}
```

Figure 100: Appending data to a `Vec`

use N bytes, where $M < N$, we must: (1) allocate a new area of N bytes; (2) copy M bytes from the old area to the new area; and (3) delete the old area. This operation requires time $\mathcal{O}(M)$.

If we add one element, or in fact any ***constant*** number of elements each time a `Vec` grows, the performance of `push_back` will be quadratic.[38] For example, if we add one element at each call of `grow`, we will copy 1, then 2, then 3 elements. To get 4 elements into the `Vec`, we will have to copy $1 + 2 + 3 = 6$ elements. To get N elements into the `Vec`, we will have to copy $\frac{1}{2}N(N-1)$ elements.

If, instead, we ***multiply*** the size of the `Vec` by a constant factor, the time spent copying falls to $\mathcal{O}(N \log N)$, which is much better. The implementation of `grow` in Figure 100 uses this technique with a constant factor of 2.

If we measure the time taken to expand an array one element at a time by calls to `push_back`, we will notice that most calls are fast ($\mathcal{O}(1)$) but a few calls (when reallocation is done) are slow ($\mathcal{O}(N)$). We account for this by defining the ***amortized time complexity*** for `push_back`, which is the time for n calls divided by n, as $n \to \infty$. The amortized time complexity for our version of `push_back` is $\mathcal{O}(\log N)$.

[38] This explains the catastrophic performance of Java programs that misuse the `+` operator for strings.

9.1.3 Refactoring

We can improve the clarity of the implementation of `Vec` by doing some simple refactoring. ***Refactoring*** means rearranging code to improve its maintainability or performance without changing its functionality. In this case, we introduce two overloaded versions of a private function called `create` to manage the creation of new `Vec`s. We can use this function to simplify the code for the constructors and the assignment operator. The two versions of `create` are declared in
the `private` part of the class declaration: 327

```
void create(size_type n = 0, const T & val = T());
void create(const_iterator begin, const_iterator end);
```

The first version has two parameters, for the size of the array and the initial values, respectively. Both parameters have default values, so that calling `create()` yields an empty `Vec`. The second version also has two parameters that must be iterators defining a range of some other compatible `Vec`.

The function `create` is responsible for allocating memory and for initializing the pointers `data`,
`avail`, and `limit`. Figure 101 shows the implementations of both versions. Figure 102 on the 328
next page shows the revised definitions of functions that use `create`. Finally, Figure 103 on 329
page 185 shows the declaration of class `Vec` with all the changes that we have discussed. 330

331

332

```
template<typename T>
void Vec<T>::create(size_type n = 0, const T & val = T())
{
    data = new T[n];
    avail = data + n;
    limit = avail;
    for (iterator p = data; p != avail; ++p)
        *p = val;
}

template<typename T>
void Vec<T>::create(const_iterator begin, const_iterator end)
{
    size_type size = end - begin;
    data = new T[size];
    avail = data + size;
    limit = avail;
    for (iterator p = data; p != avail; ++p, ++begin)
        *p = *begin;
}
```

Figure 101: Implementation of two overloads of `create`

```
template <typename T>
Vec<T>::Vec<T>()
{
    create();
}

template <typename T>
Vec<T>::Vec<T>(size_t n, const T & val)
{
    create(n, val);
}

template <typename T>
Vec<T>::Vec<T>(const Vec<T> & other)
{
    create(other.begin(), other.end());
}

template <typename T>
Vec<T> & Vec<T>::operator=(const Vec<T> & rhs)
{
    if (&rhs != this)
    {
        delete [] data;
        create(rhs.begin(), rhs.end());
    }
    return *this;
}
```

Figure 102: Revised definitions of functions that use `create`

9.2 A Note on Iterators

It might seem from the example of class `Vec` that an iterator is just a pointer and that we can use `++`, `==`, and so on, just because these functions are defined for pointers.

In fact, this is not always the case. The STL class `list`, for example, stores data in a linked list of nodes. An iterator value identifies a node. Incrementing the iterator means moving it to the
333 next node. As Figure 104 on page 186 shows, achieving this requires defining all of the iterator
334 functions, including `*` and `->`.

335 In particular, note that the prefix operators `--it` and `++it` are more efficient than the postfix operators `it--` and `it++`, because the postfix operators use the copy constructor to create the result.

```
template <typename T>
class Vec
{
public:
    typedef T* iterator;
    typedef const T* const_iterator;
    typedef size_t size_type;
    typedef T value_type;
    Vec();
    explicit Vec(size_t n, const T & val = T());
    Vec(const Vec & other);
    ~Vec();

    Vec & operator=(const Vec & rhs);

    T & operator[](size_type i) { return data[i]; }
    const T & operator[](size_type i) const { return data[i]; }

    size_type size() const { return avail - data; }

    iterator begin() { return data; }
    const_iterator begin() const { return data; }

    iterator end() { return avail; }
    const_iterator end() const { return avail; }

    void push_back(const T & val);

private:
    void create(size_type n = 0, const T & val = T());
    void create(const_iterator begin, const_iterator end);
    void grow();
    iterator data;
    iterator avail;
    iterator limit;
};
```

Figure 103: Declaration for class `Vec`: version 2

```
const_reference operator*() const
    {    // return designated value
    return (_Myval(_Ptr));
    }

_Ctptr operator->() const
    {    // return pointer to class object
    return (&**this);
    }

const_iterator& operator++()
    {    // preincrement
    _Ptr = _Nextnode(_Ptr);
    return (*this);
    }

const_iterator operator++(int)
    {    // postincrement
    const_iterator _Tmp = *this;
    ++*this;
    return (_Tmp);
    }

const_iterator& operator--()
    {    // predecrement
    _Ptr = _Prevnode(_Ptr);
    return (*this);
    }

const_iterator operator--(int)
    {    // postdecrement
    const_iterator _Tmp = *this;
    --*this;
    return (_Tmp);
    }

bool operator==(const const_iterator& _Right) const
    {    // test for iterator equality
    return (_Ptr == _Right._Ptr);
    }

bool operator!=(const const_iterator& _Right) const
    {    // test for iterator inequality
    return (!(*this == _Right));
    }
```

Figure 104: An extract from the STL class `list`

9.3 Templates vs. Inheritance: Tree traversal

In this section, we discuss an example built on the following observation: we can define a general traversal routine for a tree using a generic store, and then specialize the traversal by providing different kinds of store.

For simplicity, we will use binary trees, although the technique can be extended to general trees
and graphs. Figure 105 shows pseudocode for the general traversal. In this pseudocode, we 336
assume that we are given the `root` of a binary tree and an empty `store`. The `store` provides operations `put` and `get`, and a test `empty`. The order of the traversal is determined by the behaviour of the store. If the store is a stack with LIFO behaviour, we obtain depth-first search; if the store is a queue with FIFO behaviour, we obtain breadth-first search. If we have some knowledge about the nodes, we can obtain smarter traversals by using, for example, a priority queue as the store.

```
store.put(root);
while (!store.empty())
{
    node = store.get();
    node.visit();
    if (node.right != 0)
        store.put(node.right);
    if (node.left != 0)
        store.put(node.left);
}
```

Figure 105: Pseudocode for binary tree traversal

The precise requirements of the store are as follows:

- `put` must add a new element to the store.
- `empty` must return `true` iff the store is empty and `false` otherwise. `empty` must return `false` if there is any element that has been `put` into the store but has not been retrieved by `get`.
- `get` must return an element from the store unless the store is empty, in which case its behaviour is undefined.

The goal, then, is to implement a traversal function that is passed a store as argument and implements the traversal without knowing precisely how the store behaves. There are two obvious ways in which we might implement a generic traversal function: using inheritance and using templates. Before we show how to do this, we will look at the various features the program needs. All of the classes are intended to have just enough complexity to support this particular application. To avoid confusion, they use simple, pointer-based techniques rather than STL features.

First, we introduce the binary trees that are to be traversed: see Figure 106 on page 189. There 337
is no class for complete trees, just a class for nodes of trees. Each node has an integer key and 338
pointers to its left and right subtrees. Either or both of these pointers, of course, may be null. 339

There are two functions associated for binary trees. One of them overloads `operator<<` and is used as a check to see what is in the tree. The other, `makeTree`, constructs a tree containing all of the keys in a given semi-closed range.

For both kinds of store (stack and queue), we will use a single-linked list to store items, as shown
340 in Figure 107 on page 190. The main difference is that a stack accesses only one end of the list (the "top" of the stack) and the queue accesses both ends (the "front" and "back" of the queue). We have assumed that list items are pointers to tree nodes but, of course, we could generalize this by making `ListNode` a template class.

Stores are implemented with an abstract base class with pure virtual functions for the required operations: see Figure 108 on page 190.

341 Figure 109 on page 191 shows the declaration of class `Stack`. To save space, the member functions
342 are defined in the body of the class declaration. (This is allowed, and has the side-effect of permitting the compiler to inline them, as we will discuss later.)

Pushing an element onto the stack (`put()`) is easy: we just have to create a new list node and update the pointer to the head of the list, which represents the top of the stack.

Retrieving an element from the stack is slightly harder and care is necessary. We have to get the pointer from the first list node, delete the list node, and update the `top` pointer. This requires creating temporaries, `result` and `tmp`, to store the node to be returned and the new top pointer while we delete the old one. The assertion will detect an attempt to pop an element from an empty stack, although this should never happen: in the traversal pseudocode of Figure 105 on the previous page, we call `get` only after checking that the store is not empty.

343 Figure 110 on page 192 shows the declaration of class `Queue`. The code is somewhat more
344 complicated than the stack code because we have to maintain two pointers, to the front and back of the list.

Points to note about class `Queue`:

- The list pointers run from the front of the queue to the back of the queue.
- New items are inserted at the back of the list. The `back` pointer is updated to point to the new item.
- Items are removed from the front of the list. The `front` pointer is updated to point to the next item in the list.
- If the list is empty, `front` = `back` = 0. When an item is inserted into an empty queue, both `front` and `back` pointers are set pointing to it.
- If removing an item makes the `front` pointer null, then the `back` pointer must be set to null as well, and the queue is then empty.
- The test `empty` examines the `back` pointer, although it could in fact examine either pointer.
- An undocumented class invariant: ***either the front and back pointers are both null, or both are non-null.***

345 At this point, we have all of the code required for the demonstration. Figure 111 on page 193
shows a generic traversal function that is passed a pointer to the abstract base class `Store`. This
346 function can be invoked either by calling

```
// A class for binary trees with integer nodes.
class TreeNode
{
public:
    TreeNode(int key, TreeNode *left, TreeNode *right);
    ~TreeNode();
    int getKey() { return key; }
    TreeNode *getLeft() { return left; }
    TreeNode *getRight() { return right; }
private:
    int key;
    TreeNode *left;
    TreeNode *right;
};

TreeNode::TreeNode(int key, TreeNode *left, TreeNode *right)
    : key(key), left(left), right(right)
{}

TreeNode::~TreeNode()
{
    delete left;
    delete right;
}

// Display a tree using inorder traversal.
ostream & operator<<(ostream & os, TreeNode *ptn)
{
    if (ptn != 0)
    {
       if (ptn->getLeft() != 0)
           os << ptn->getLeft();
       os << ptn->getKey() << ' ';
       if (ptn->getRight() != 0)
           os << ptn->getRight();
    }
    return os;
}

// Construct a tree with keys in [first,last).
TreeNode * makeTree(int first, int last)
{
    if (first == last)
        return 0;
    else
    {
        int mid = (first + last) / 2;
        return new TreeNode(mid, makeTree(first, mid), makeTree(mid + 1, last));
    }
}
```

Figure 106: Binary tree: class declaration and related functions

```
class ListNode
{
public:
    ListNode(TreeNode *ptn, ListNode *next) : ptn(ptn), next(next) {}
    TreeNode *ptn;
    ListNode *next;
};
```

Figure 107: Class `ListNode`

```
class Store
{
public:
    virtual void put(TreeNode *ptn) = 0;
    virtual TreeNode *get() = 0;
    virtual bool empty() = 0;
};
```

Figure 108: Abstract base class `Store`

```
traverseUsingInheritance(tree, new Stack);
```

or by calling

```
traverseUsingInheritance(tree, new Queue);
```

The decision as to which kind of store to use is made at ***run-time***. Each call to `put`, `get`, or `empty` is dynamically bound to the corresponding function in either `Stack` or `Queue`. The overhead of dynamic binding is small but could be significant if these operations are performed
347 very often.

Figure 112 on page 193 shows a traversal function that obtains its genericity with a template. The template parameter `StoreType` must be replaced by a class that provides the operations `put`, `get`, and `empty`. It must also provide a default constructor that creates an empty store. This constructor is invoked by the statement

```
StoreType *pst = new StoreType;
```

The inheritance version does not need this statement because it is passed an empty store. Except for this statement, the bodies of the two functions are identical.

348 The template function is invoked either by calling

```
traverseUsingTemplates<Stack>(tree);
```

or by calling

```
class Stack : public Store
{
public:
    Stack() : top(0) {}

    void put(TreeNode *ptn)
    {
        top = new ListNode(ptn, top);
    }

    TreeNode *get()
    {
        assert(top);
        TreeNode *result = top->ptn;
        ListNode *tmp = top->next;
        delete top;
        top = tmp;
        return result;
    }

    bool empty()
    {
        return top == 0;
    }

private:
    ListNode *top;
};
```

Figure 109: Class `Stack`

```
traverseUsingTemplates<Queue>(tree);
```

Note that we must provide the template argument `<Stack>` or `<Queue>` explicitly in calls to `traverseUsingTemplates`, because the compiler cannot infer which kind of store we want from the function argument `tree`. The call 349

```
traverseUsingTemplates(tree);
```

produces the error message (quite intuitively, for a change):

```
error C2783: 'void traverseUsingTemplates(TreeNode *)' :
   could not deduce template argument for 'StoreType'
```

350

Figure 113 on page 194 shows a program that tests the generic traversal, using both the inheritance and the template versions, and the output from this program. 351

Comparison of the solutions:

```
class Queue : public Store
{
public:
    Queue() : front(0), back(0)
    {}

    void put(TreeNode *ptn)
    {
        ListNode *tmp = new ListNode(ptn, 0);
        if (back == 0)
            front = tmp;
        else
            back->next = tmp;
        back = tmp;
    }

    TreeNode *get()
    {
        assert(front);
        TreeNode *result = front->ptn;
        ListNode *tmp = front->next;
        delete front;
        front = tmp;
        if (front == 0)
            back = 0;
        return result;
    }

    bool empty()
    {
        return back == 0;
    }

private:
    ListNode *front;
    ListNode *back;
};
```

Figure 110: Class `Queue`

```
void traverseUsingInheritance(TreeNode *root, Store *pst)
{
    pst->put(root);
    while (!pst->empty())
    {
        TreeNode *ptn = pst->get();
        assert(ptn);
        cout << ptn->getKey() << ' ';
        if (ptn->getRight() != 0)
            pst->put(ptn->getRight());
        if (ptn->getLeft() != 0)
            pst->put(ptn->getLeft());
    }
}
```

Figure 111: Generic traversal using inheritance

```
template<typename StoreType>
void traverseUsingTemplates(TreeNode *root)
{
    StoreType *pst = new StoreType;
    pst->put(root);
    while (!pst->empty())
    {
        TreeNode *ptn = pst->get();
        assert(ptn);
        cout << ptn->getKey() << ' ';
        if (ptn->getRight() != 0)
            pst->put(ptn->getRight());
        if (ptn->getLeft() != 0)
            pst->put(ptn->getLeft());
    }
}
```

Figure 112: Generic traversal using templates

- For inheritance to work, the classes `Stack` and `Queue` must be derived from the same base class. The traversal function has a parameter of type pointer-to-base-class.

  The template solution does not require any relationship between classes `Stack` and `Queue`. The only requirement is that each class has a default constructor that creates an empty store and implements the required member functions.

- The solutions are a trade-off between compile-time overhead and run-time overhead. Dynamic binding has a small performance penalty because virtual functions are called indirectly (through a pointer) rather than directly.

  Templates have no run-time overhead but it does slow down compilation, sometimes

```
int main()
{
    TreeNode *tree = makeTree(0, 10);
    cout << "Initial tree (in order): " << tree << endl;

    cout << endl << "Traverse using inheritance with Stack: ";
    traverseUsingInheritance(tree, new Stack);
    cout << endl << "Traverse using inheritance with Queue: ";
    traverseUsingInheritance(tree, new Queue);
    cout << endl;

    cout << endl << "Traverse using templates with Stack: ";
    traverseUsingTemplates<Stack>(tree);
    cout << endl << "Traverse using templates with Queue: ";
    traverseUsingTemplates<Queue>(tree);
    cout << endl;

    delete tree;
    return 0;
}
```

Output:

```
Initial tree (in order): 0 1 2 3 4 5 6 7 8 9

Traverse using inheritance with Stack: 5 2 1 0 4 3 8 7 6 9
Traverse using inheritance with Queue: 5 8 2 9 7 4 1 6 3 0

Traverse using templates with Stack: 5 2 1 0 4 3 8 7 6 9
Traverse using templates with Queue: 5 8 2 9 7 4 1 6 3 0
```

Figure 113: Test program and output for generic tree traversal

significantly. A small amount of time is spent expanding templates. Much more time is wasted reading additional header files, because the bodies of template functions must be put in header files rather than implementation files.

> *"This is a real problem in practice because it considerably increases the time needed by the compiler to compile significant programs.."* (Vandevoorde, Josuttis, and Gregor 2017)

- As a general rule, early binding provides efficiency and late binding provides flexibility. With both versions, we can use both types of store, as the test program generates. However,
352 we may have to make a ***dynamic*** choice of store, as shown in the following code:

```
Store *ps;
if ( ⟨condition⟩ )
```

```
        ps = new Stack;
    else
        ps = new Queue;
    traverseUsingInheritance(tree, ps);
```

or perhaps like this:

```
    traverseUsingInheritance(tree, ⟨condition⟩ ? new Stack : new Queue);
```

If this kind of code cannot be avoided (i.e., the value of ⟨*condition*⟩ must be evaluated at run-time), there is no reasonable alternative to inheritance. The alternative 353

```
    if ( ⟨condition⟩ )
        traverseUsingTemplates<Stack>(tree);
    else
        traverseUsingTemplates<Queue>(tree);
```

is not particularly attractive. It forces the compiler to generate two expansions of the traversal function, which will occupy lots of memory, and the gain in performance will probably be insignificant.

- In this example, the base class `Store` contains no code. In a larger, more practical, application, code duplicated in derived classes could be moved into the base class.

  Common code cannot be as easily shared in the template version. However, it is possible to use the template approach with a class hierarchy and thereby to obtain the benefits of code sharing. The difference, as before, is just that the inheritance version chooses the derived class at run-time but the template version makes the choice at compile-time.

- In terms of maintenance, there is not a lot to choose between the two solutions. An advantage of the inheritance version is that a new derived class can be added to the hierarchy and linked in to the program without recompiling the traversal function. This might be useful in a large-scale project.

9.3.1 Organizing the code

354

Figure 114 on the following page shows the organization of the various components of the traversal program, not including the `main` function. The header and implementation files for `TreeNode` are conventional. The header file `store.h` contains the forward declaration 355 356 forward declaration

```
class ListNode;
```

It does not need to contain the complete declaration of class `ListNode` because all of the references to `ListNode` are pointers (or references). Provided that the compiler is informed that `ListNode` is a class, it does not need to know how big an instance is (we will discuss forward declarations in more detail later).

The implementation file `store.cpp` contains the full declaration of `ListNode`. Since it is in an implementation file, this declaration is inaccessible to other parts of the program. That its members are all public does not matter, because they are only available to the class `Stack` and `Queue`.

There is no implementation for class `Store` because it is an abstract class that contains only pure virtual functions.

binarytree.h

Declarations of class `TreeNode`,
free function `makeTree`,
`operator<<` for `TreeNode`s.

store.h

`#include "binarytree.h"`

Forward declaration for class `ListNode`; declarations for classes `Store`, `Stack`, and `Queue`.

binarytree.cpp

`#include "binarytree.h"`

Implementations of `TreeNode` functions, `makeTree`, and `operator<<`.

store.cpp

`#include "store.h"`

Declaration of class `ListNode`, implementation of classes `Stack` and `Queue`.

traversal.cpp

`#include "binarytree.h"`
`#include "store.h"`

Implementations of `traverseUsingInheritance`, `traverseUsingTemplates`, and `main`.

Figure 114: Code organization for traversal program

9.4 The Curiously Recurring Template Pattern

The Curiously Recurring Template Pattern (CRTP) was named by James "Cope" Coplien
357 (Coplien 1995). Figure 115 shows the basic idea. At first sight, it is rather mysterious. An example may clarify things.

```
class X : public Base<X>
{
    ....
};
```

Figure 115: The Curiously Recurring Template Pattern

Consider this problem: we would like to provide a way of extending any class that provides `operator<` so that it also provides `operator>`.

358 We might try to apply inheritance in the usual way. We define a base class:

```
class Ordered
{
```

```
        virtual bool operator<(const Ordered & other) = 0;
        bool operator>(const Ordered & other)
        {
            return other < *this;
        }
    }
```

Then we can inherit this base class to give the desired functionality:

```
class Widget : public Ordered
{
    bool operator<(const Widget & other) { .... }
    ....
};
```

Unfortunately, this doesn't work. The redefinition of `operator<` in class `Widget` is incorrect because the parameter type is different. Moreover, `Ordered::operator>` provides an instance of `Ordered` for comparison, but we want to compare another `Widget`.

```
template <class T>
class Ordered
{
public:
    bool operator>(const T & rhs) const
    {
        const T & self = static_cast<const T &>(*this);
        return rhs < self;
    }
};
```

Figure 116: A class that provides `operator>` using `operator<`

CRTP comes to the rescue! We define the base class `Ordered` as a template class, as shown
in Figure 116. We have to cast from `Ordered` to `T`, but we will see that this doesn't matter in 359
practice. The revised version of class `Widget` inherits from a version of class `Ordered` that is customized for `Widget`s, using the "recursive" template argument:

```
class Widget : public Ordered<Widget>
{
    bool operator<(const Widget & other) { .... }
    ....
};
```

After templates have been expanded, we have a class that looks something like this (or rather,
would look like this if we could see it): 360

```
class OrderedWidget
{
public:
    bool operator>(const Widget & rhs) const
    {
        const Widget & self = static_cast<const Widget &>(*this);
        return rhs < self;
    }
};
```

We can now see that the cast is perfectly harmless, because it is just casting a `const Widget &` to itself!

We could have solved this problem more easily with templates, as is done in the STL namespace `rel_ops`: see Figure 117. But, as another example of CRTP, consider the problem of keeping track of the number of instances of a class. This is easily done by providing a `static` counter in the class. But suppose we want to encapsulate this behaviour, providing a base class whose children can all keep track of their instances.

```
namespace std
{
  namespace rel_ops
  {
    template <class _Tp>
      inline bool
      operator!=(const _Tp& __x, const _Tp& __y)
      { return !(__x == __y); }

    template <class _Tp>
      inline bool
      operator>(const _Tp& __x, const _Tp& __y)
      { return __y < __x; }

    template <class _Tp>
      inline bool
      operator<=(const _Tp& __x, const _Tp& __y)
      { return !(__y < __x); }

    template <class _Tp>
      inline bool
      operator>=(const _Tp& __x, const _Tp& __y)
      { return !(__x < __y); }

  } // namespace rel_ops
} // namespace std
```

Figure 117: Defining relational operators with templates in the STL

The obvious way of doing this doesn't work. If we put a **static** counter in the base class, we will count *all* instances of child classes. We need a way of providing a separate counter for *each* child class. Once again, CRTP shows the way. Figure 118 shows the base class. 361

```
template<class T>
class Counted
{
public:
   Counted() { ++counter; }
   ~Counted() { --counter; }
   static long num() { return counter; }
private:
   static long counter;
};
```

Figure 118: Base class `Counted`

To make class `T` a "counted" class, it must inherit from `Counted<T>`. Whenever we create such a class, we must also provide an initialized definition for its counter:

```
class Widget : public Counted<Widget> { .... };
long Counted<Widget>::counter = 0;
```

We can create as many counted classes as we need:

```
class Goblet : public Counted<Goblet> { .... };
long Counted<Goblet>::counter = 0;
```

362

Figure 119 on the next page shows a program that tests these ideas. The output shown in Figure 120 on the following page demonstrates that the counts are maintained correctly as objects are created and destroyed. 363

Note that the destructor is ***not*** virtual. We don't have to make it virtual, since CRTP does not use polymorphism, even though it makes use of inheritance. This is a concrete example of a programming technique known as ***Mixin***. In general, this technique allows you to add features to a class without inheriting from a base class. The class that provides the new (typically a single) feature, like counting in the example above, is the Mixin.[39] The CRTP idiom is also sometimes referred to as ***F-bound polymorphism*** or ***Upside-Down Inheritance***. For more on CRTP, see (Vandevoorde, Josuttis, and Gregor 2017, Chapter 21.2). Mixin

References

Coplien, J. O. (1995). Curiously recurring template patterns. *C++ Report 7*(2), 24–27.

Koenig, A. and B. E. Moo (2000). *Accelerated C++: Practical Programming by Example.* Addison-Wesley.

[39] Java 8 added default method implementations in interfaces, which can be seen as a way of providing Mixins.

```
void report(string title)
{
   cout << title << endl <<
      "Widgets: " << Widget::num() << endl <<
      "Goblets: " << Goblet::num() << endl << endl;
}

int main()
{
   Widget w1;
   Goblet g1;
   Goblet g2;
   {
      Widget w2;
      Widget w3;
      Goblet g3;
      report("Inner:");
   }
   report("Outer:");
   return 0;
}
```

Figure 119: Testing the `Counted` hierarchy

```
Inner:
Widgets: 3
Goblets: 3

Outer:
Widgets: 1
Goblets: 2
```

Figure 120: Output from the counting test of Figure 119

Meyers, S. (2005). *Effective C++: 55 Specific Ways to Improve Your Programs and Designs* (3rd ed.). Addison-Wesley.

Vandevoorde, D., N. M. Josuttis, and D. Gregor (2017). *C++ Templates* (2nd ed.). Addison-Wesley. http://tmplbook.com.