deployed, however, it is entirely possible that the procedure could be applied to strings of over one million characters. All of a sudden this benign piece of code has become a major performance bottleneck. By contrast, the performance of `lower2` will be adequate for strings of arbitrary length. Stories abound of major programming projects in which problems of this sort occur. Part of the job of a competent programmer is to avoid ever introducing such asymptotic inefficiency.

Practice Problem 5.3 (solution page 609)

Consider the following functions:

```
long min(long x, long y) { return x < y ? x : y; }
long max(long x, long y) { return x < y ? y : x; }
void incr(long *xp, long v) { *xp += v; }
long square(long x) { return x*x; }
```

The following three code fragments call these functions:

A.
```
for (i = min(x, y); i < max(x, y); incr(&i, 1))
    t += square(i);
```

B.
```
for (i = max(x, y) - 1; i >= min(x, y); incr(&i, -1))
    t += square(i);
```

C.
```
long low = min(x, y);
long high = max(x, y);

for (i = low; i < high; incr(&i, 1))
    t += square(i);
```

Assume `x` equals 10 and `y` equals 100. Fill in the following table indicating the number of times each of the four functions is called in code fragments A–C:

Code	min	max	incr	square
A.				
B.				
C.				

5.5 Reducing Procedure Calls

As we have seen, procedure calls can incur overhead and also block most forms of program optimization. We can see in the code for `combine2` (Figure 5.6) that `get_vec_element` is called on every loop iteration to retrieve the next vector element. This function checks the vector index `i` against the loop bounds with every vector reference, a clear source of inefficiency. Bounds checking might be a useful feature when dealing with arbitrary array accesses, but a simple analysis of the code for `combine2` shows that all references will be valid.

code/opt/vec.c

```
data_t *get_vec_start(vec_ptr v)
{
    return v->data;
}
```

code/opt/vec.c

```
/* Direct access to vector data */
void combine3(vec_ptr v, data_t *dest)
{
    long i;
    long length = vec_length(v);
    data_t *data = get_vec_start(v);

    *dest = IDENT;
    for (i = 0; i < length; i++) {
        *dest = *dest OP data[i];
    }
}
```

Figure 5.9 Eliminating function calls within the loop. The resulting code does not show a performance gain, but it enables additional optimizations.

Suppose instead that we add a function `get_vec_start` to our abstract data type. This function returns the starting address of the data array, as shown in Figure 5.9. We could then write the procedure shown as `combine3` in this figure, having no function calls in the inner loop. Rather than making a function call to retrieve each vector element, it accesses the array directly. A purist might say that this transformation seriously impairs the program modularity. In principle, the user of the vector abstract data type should not even need to know that the vector contents are stored as an array, rather than as some other data structure such as a linked list. A more pragmatic programmer would argue that this transformation is a necessary step toward achieving high-performance results.

			Integer		Floating point	
Function	Page	Method	+	*	+	*
`combine2`	545	Move `vec_length`	7.02	9.03	9.02	11.03
`combine3`	549	Direct data access	7.17	9.02	9.02	11.03

Surprisingly, there is no apparent performance improvement. Indeed, the performance for integer sum has gotten slightly worse. Evidently, other operations in the inner loop are forming a bottleneck that limits the performance more than the call to `get_vec_element`. We will return to this function later (Section 5.11.2) and see why the repeated bounds checking by `combine2` does not incur a performance penalty. For now, we can view this transformation as one of a series of steps that will ultimately lead to greatly improved performance.

5.6 Eliminating Unneeded Memory References

The code for combine3 accumulates the value being computed by the combining operation at the location designated by the pointer dest. This attribute can be seen by examining the assembly code generated for the inner loop of the compiled code. We show here the x86-64 code generated for data type double and with multiplication as the combining operation:

```
Inner loop of combine3.  data_t = double, OP = *
dest in %rbx, data+i in %rdx, data+length in %rax
.L17:                                  loop:
  vmovsd   (%rbx), %xmm0                 Read product from dest
  vmulsd   (%rdx), %xmm0, %xmm0          Multiply product by data[i]
  vmovsd   %xmm0, (%rbx)                 Store product at dest
  addq     $8, %rdx                      Increment data+i
  cmpq     %rax, %rdx                    Compare to data+length
  jne      .L17                          If !=, goto loop
```

We see in this loop code that the address corresponding to pointer dest is held in register %rbx. It has also transformed the code to maintain a pointer to the *i*th data element in register %rdx, shown in the annotations as data+i. This pointer is incremented by 8 on every iteration. The loop termination is detected by comparing this pointer to one stored in register %rax. We can see that the accumulated value is read from and written to memory on each iteration. This reading and writing is wasteful, since the value read from dest at the beginning of each iteration should simply be the value written at the end of the previous iteration.

We can eliminate this needless reading and writing of memory by rewriting the code in the style of combine4 in Figure 5.10. We introduce a temporary variable acc that is used in the loop to accumulate the computed value. The result is stored at dest only after the loop has been completed. As the assembly code that follows shows, the compiler can now use register %xmm0 to hold the accumulated value. Compared to the loop in combine3, we have reduced the memory operations per iteration from two reads and one write to just a single read.

```
Inner loop of combine4.  data_t = double, OP = *
acc in %xmm0, data+i in %rdx, data+length in %rax
.L25:                                  loop:
  vmulsd   (%rdx), %xmm0, %xmm0          Multiply acc by data[i]
  addq     $8, %rdx                      Increment data+i
  cmpq     %rax, %rdx                    Compare to data+length
  jne      .L25                          If !=, goto loop
```

We see a significant improvement in program performance, as shown in the following table: