Functional Programming with C++

CHRIS WEED

Table of Contents

Functional Programming with C++

An Alternative Perspective on C++

By Chris Weed

Chris Weed

chrisweed@gmail.com

About the Author

Chris Weed is a software engineer who lives and works in the Boston area of Massachusetts. Chris has been building C++ software since 1995, and endeavors to use functional programming every day. He has found functional programming is key to writing understandable and less buggy software. This has resulted in greater productivity and happier software engineers. His interests include computer vision, video processing, software patterns, and rapid development.

Additional Titles

Introduction

Overview

Functional programming is experiencing a resurgence with languages like Python, Haskell, and Scala. C++ and Java have also added functional features, such as lambdas and futures. Writing C++ in a functional style using const variables, functions without side effects, recursive functions, and function objects results in code that is often simpler and easier to maintain and understand, especially when poorly documented. This book explores functional techniques in C++ code and their advantages and disadvantages.

Side effects

One of the most important features of functional programming is that programs are written almost exclusively without side effects. According to Wikipedia, a function has a **side effect** if, in addition to returning a value, it also modifies some state or has an *observable* interaction with calling functions or the outside world. For example, a function might modify a global variable or static variable, modify one of its arguments, raise an exception, write data to a display or file, read data, or call other side-effecting functions. [1]

Functions without side effects are generally referred to as “pure” functions, and are the norm for functional languages. These functions only produce output variables based on the input arguments. However, C++ code commonly includes functions with mutable arguments, references global, static, or member variables, or calls other non-pure functions.

A useful property of pure functions is that when called with the same arguments always results in the same output. Due to this, functional languages can cache the reproducible result using a process called “memoization” for following calls to pure functions. A non-pure function is based on some external state or changes an external state, which is often difficult to reproduce, test, or understand.

Sequences

STL algorithms are generally based on iterating over a sequence of values or objects accessed between the “begin” and “end” iterators. The typical segmentation of a list in a functional language is between the head object and tail list. STL containers provide the head object by dereferencing the “begin” iterator, and the tail list is available between the “next” iterator, accessed by std::next(begin), and the “end” iterator. Additionally, an empty list is when the “begin” and “end” iterator are equal, such that there are no objects between them. The Algorithm chapter uses this model of a sequence, extensively.

Compiler

The Apple LLVM version 6.1.0 compiler was used to compile the samples in this book using the “-O3” option that includes tail-call optimization. This optimization setting is necessary in release and debug mode builds with deep recursion, since the program will exceed the stack size without it and result in a segmentation fault. The option “-std=c++11” was also used for building all of the code samples. For code samples related to C++14, the “-std=c++14” option was used instead.

Performance

Functional programming generally involves an abstraction penalty that prevents the code from achieving the absolute optimal performance, however, that is rarely necessary for an entire program. Often, only small, performance critical sections of a program need to be optimized for good performance. Restricting side effects to those sections allows nearly optimal performance while maintaining readable and error free code throughout the remaining parts.

Algorithms

Recursive functions

Recursive functions are a staple of functional languages and many algorithms are more easily implemented than with a strictly iterative formulation. A difficulty with recursive algorithms is a computer's stack limit, which will be violated with too many recursive calls. Modern C++ compilers implement tail-call optimization, and properly structured tail-recursive functions are not limited by the stack size. The compiler generates code that does not add a stack frame and will not exceed the stack limit with a high number of recursive calls.

This chapter covers recursive implementations of STL algorithms, and most are implemented with tail-recursion. Some algorithms include non-tail-recursive implementations for exposition or when they are found to compile to code that will not exceed the stack. All of the functions in this chapter were tested with a very large recursion depth, and none resulted in a segmentation fault.

Most of the algorithms covered in this chapter are non-modifying algorithms that will not modify the data processed by the algorithm. These algorithms can process data accessed by const iterators that transform the data without changing it. Iterative implementations of these algorithms are available for comparison at the website http://cplusplus.com here:

<algorithm> http://www.cplusplus.com/reference/algorithm/

<numeric>

http://www.cplusplus.com/reference/numeric/

all_of

The all_of algorithm returns true if all of the elements of a sequence are true when individually passed to the unary predicate. This algorithm checks each element until the predicate returns false or reaches the end of the sequence. Since the AND operator (&&) short-circuits evaluation if the argument to the left of it is false, the all_of function on the right side of it will not be called and the recursion will stop.

recursive::all_of

```
template<class InputIterator,
         class UnaryPredicate>
bool all_of (InputIterator first,
             InputIterator last,
             UnaryPredicate pred)
{
   if (first == last) return true;
   return pred(*first)) &&
     all_of(std::next(first), last, pred);
}
```

Most of the algorithms in this chapter start with a conditional statement that checks whether the end of the sequence has been reached and the first and last iterator are equal. It is followed by the logic implemented by the algorithm. The following, tail recursive version uses an additional if-conditional for checking the predicate applied to the first element.

recursive::all_of

```
template<class InputIterator,
         class UnaryPredicate>
bool all_of (InputIterator first,
             InputIterator last,
             UnaryPredicate pred)
{
   if (first == last) return true;
   if (!pred(*first)) return false;
   return all_of(std::next(first), last, pred);
}
```

Neither of these implementations resulted in a segmentation fault from exceeding the stack, while the first one is more succinct. An implementation with only a single return statement is possible using the OR operator (||).

recursive::all_of

```
template<class InputIterator,
         class UnaryPredicate>
bool all_of (InputIterator first,
             InputIterator last,
             UnaryPredicate pred)
{
  return
    (first == last) ||
    pred(*first)) &&
    all_of(std::next(first), last, pred);
}
```

This single statement captures the recursive tail condition in the algorithm calculation.

any_of

The any_of algorithm applies a unary predicate to elements of a sequence and returns true if the predicate returns true for any of the elements. It returns false for a sequence with zero elements.

recursive::any_of

```
template<class InputIterator,
         class UnaryPredicate>
bool any_of (InputIterator first,
             InputIterator last,
             UnaryPredicate pred)
{
  if (first == last) return false;

  return pred(*first)) ||

     any_of(std::next(first), last, pred);
}
```

Due to short-circuit evalution, if the predicate returns true, the recursive call to any_of on the right side of the OR operator will not be called. The following, tail recursive version reproduces the short-circuit evaluation and returns before the recursive call if the predicate is true.

recursive::any_of

```
template<class InputIterator,
         class UnaryPredicate>
bool any_of (InputIterator first,
             InputIterator last,
             UnaryPredicate pred)
{
  if (first == last) return false;

  if (pred(*first)) return true;

  return any_of(std::next(first), last, pred);
}
```

Neither of these implementations resulted in a segmentation fault from exceeding the stack, while the first one seems more succinct.

none_of

The none_of algorithm returns true if the predicate is false for all of the elements of the sequence or the sequence is empty. Otherwise, the algorithms returns false. The algorithm is almost identical to the all_of algorithm, but takes the negation of the predicate.

recursive::none_of

```
template<class InputIterator,
         class UnaryPredicate>
bool none_of(
  InputIterator first,
  InputIterator last,
  UnaryPredicate pred)
{
  if (first == last) return true;

  return !pred(*first) &&
    none_of(std::next(first), last, pred);

}
```

The none_of algorithms is implementable with the all_of algorithm and a lambda that negates the predicate. Lambda’s are discussed at length in a later chapter, and in this case it is used to create the lambda passed to the all_of function.

recursive::none_of

```
template<class InputIterator,
         class UnaryPredicate>
bool none_of (InputIterator first,
             InputIterator last,
             UnaryPredicate pred)
{
  return
    all_of(std::next(first),
           last,
           [&](auto x){ return !pred(x); });

}
```

The lambda is a C++14-style generic lambda with an “auto” argument type, and it is possible to use a C++11-style lambda with the argument type being “typename std::iterator_traits<InputIterator>::value_type”.

for_each

The for_each algorithm applies a function object to every element of a sequence, and returns the function object as the result.

recursive::for_each

```
template<class InputIterator,
         class Function>
Function
for_each(
  InputIterator first,
  InputIterator last,
  Function fn)
{
  if (first == last) return fn;
  fn(*first);
  return for_each(std::next(first), last, fn);
}
```

Returning the function object from for_each is likely only valuable for the case where the function object changes it’s state, since an object that is constant would merely be the same as the function object passed in.

find

The find algorithm returns an iterator to the first element that is equal to the passed-in value. If an element is not found the last iterator is returned.

recursive::find

```
template<class InputIterator,
         class T>
InputIterator
find (InputIterator first,
      InputIterator last,
      const T& val)
{
  if (first == last) return first;
  if (*first == val) return first;
  return find(std::next(first), last, val);
}
```

The last two lines of the algorithm are implemented with the ternary operator in a single line in the following listing. This makes the algorithm more succinct.

recursive::find

```
template<class InputIterator,
         class T>
InputIterator
find (InputIterator first,
      InputIterator last,
      const T& val)
{
  if (first == last) return first;
  return (*first == val) ? first :
    find(std::next(first), last, val);
}
```

The ternary operator (? :) simplifies checking a conditional before the algorithm recurses, and is useful for simplifying more of the algorithms in this chapter. The first line can also be combined with the following line using the ternary operator to form a single line implementation.

recursive::find

```
template<class InputIterator,
         class T>
InputIterator
find(
  InputIterator first,
  InputIterator last,
  const T& val)
{
  return (first == last) ? first :
         (*first == val) ? first :
    find(std::next(first), last, val);
}
```

find_if

The find_if algorithm applies a unary predicate to each element of the sequence and returns an iterator to the first element for which the predicate returns true. The algorithm returns the last iterator, if no elements are found for which the predicate is true.

recursive::find_if

```
template<class InputIterator,
         class UnaryPredicate>
InputIterator
find_if(
  InputIterator first,
  InputIterator last,
  UnaryPredicate pred)
{
  if (first == last) return first;
  if (pred(*first)) return first;
  return find_if(std::next(first), last, pred);
}
```

This algorithm is also implementable in fewer lines with the ternary operator.

find_if_not

The find_if_not algorithm applies a unary predicate to each element of the sequence and returns an iterator to the first element for which the predicate returns false.

recursive::find_if_not

```
template<class InputIterator,
         class UnaryPredicate>
InputIterator
find_if(InputIterator first,
        InputIterator last,
        UnaryPredicate pred)
{
  if (first == last) return first;
  if (!pred(*first)) return first;
  return
    find_if_not(std::next(first), last, pred);
}
```

The find_if_not is also implementable by calling the find_if algorithm and applying a negating lambda to the passed in predicate.

recursive::find_if_not

```
template<class InputIterator,
         class UnaryPredicate>
InputIterator
find_if_not(
  InputIterator first,
  InputIterator last,
  UnaryPredicate pred)
{
  find_if(first,
          last,
          [&](auto x){ return !pred(x); });
}
```

The lambda is a C++14-style generic lambda with an “auto” argument type, and it is possible to use a C++11-style lambda with the argument type being “typename std::iterator_traits<InputIterator>::value_type”.

find_end

The find_end algorithm finds the last occurrence of a subsequence in a sequence. The algorithm calls a helper version of the find_end function with an extra argument that passes the last occurrence where the subsequence was found so far. It starts by calling the helper function with the “last” iterator as the initial possible result. When it finds a new occurrence of the subsequence using the starts_with function, it updates the possible result.

recursive::find_end

```
template<class ForwardIterator1,

         class ForwardIterator2>

bool

starts_with(ForwardIterator1 first1,

            ForwardIterator1 last1,

            ForwardIterator2 first2,

            ForwardIterator2 last2)

{

  if (first2==last2) return true;

  if (first1==last1) return false;

  return (*first1 == *first2) &&

    starts_with(std::next(first1),

                last1,

                std::next(first2),

                last2);

}


template<class ForwardIterator1,
         class ForwardIterator2>
ForwardIterator1
find_end (ForwardIterator1 first1,
          ForwardIterator1
last1,
          ForwardIterator2 first2,
          ForwardIterator2 last2,
          ForwardIterator1 curr)
```

```
{
  if (first1==last1) return curr;
  if (starts_with(first1,last1,first2,last2))
    return find_end(std::next(first1),
                    last1,
                    first2,
                    last2,
                    first1);
  return find_end(std::next(first1),
                  last1,
                  first2,
                  last2,
                  curr);
}


template<class ForwardIterator1,
         class ForwardIterator2>
ForwardIterator1
find_end(ForwardIterator1 first1,
         ForwardIterator1
last1,
         ForwardIterator2 first2,
         ForwardIterator2 last2)
{
  if (first2==last2) return last1;
  return
    find_end(first1,last1,first2,last2,last1);
}
```

The starts_with function is also later used to calculate the search algorithm.

find_first_of

The find_first_of function finds the first element in a sequence that matches one of the elements in a second sequence. The function calls the previously defined find function to match a value to the values in the second sequence.

recursive::find_first_of

```
template<class InputIterator,
         class ForwardIterator>
InputIterator
find_first_of(
  InputIterator first1,
  InputIterator last1,
  ForwardIterator first2,
  ForwardIterator last2)
{
  if (first1==last1) return last;

  if (find(first2,last2,*first1)!=last2)

    return first1;

  return find_first_of(
    std::next(first1),
    last1,
    first2,
    last2);
}
```

adjacent_find

The adjacent_find algorithm returns an iterator to the first location where two consecutive items in a sequence are equal. The iterator points to the first of the two items. If the sequence holds fewer than two items, the end iterator is returned.

The recursive implementation has a helper function to accommodate the check for an empty sequence in the main function and a check for the “next” iterator reaching the end of the sequence. The “first” and “next” iterator are used to point to the consecutive elements that are being compared.

recursive::adjacent_find

```
template <class ForwardIterator>
ForwardIterator
adjacent_find1 (ForwardIterator first,
                ForwardIterator last)

{

  ForwardIterator next = std::next(first);

  if (next == last) return last;

  if (*first == *next) return first;

  return adjacent_find1(next, last);

}


template <class ForwardIterator>
ForwardIterator
adjacent_find (ForwardIterator first,
               ForwardIterator last)
{
  if (first==last) return last;
  return adjacent_find1(first,last);
}
```

count

The count algorithm calculates the number of items in a sequence that match a particular value. This simple algorithm requires only two lines in the implementation.

recursive::count

```
template <class InputIterator,
          class T>
typename std::iterator_traits<
  InputIterator>::difference_type
count (InputIterator first,
       InputIterator last,
       const T& val)

{

  if (first==last) return 0;

  return

    (*first==val) +

    count(std::next(first), last, val);

}
```

The tail-recursive version that passes the current count to the next count call is a little more complicated. It requires calling a separate count1 function with this additional argument.

recursive::count

```
template <class InputIterator,
          class T>
typename std::iterator_traits<
  InputIterator>::difference_type
count (
  InputIterator first,
  InputIterator last,
  const T& val
  typename
    std::iterator_traits<
    InputIterator>::difference_type c = 0)
```

```
{
  if (first==last) return 0;
  return
    count1(std::next(first),
           last,
           val,
           c + (*first==val));
}


template <class InputIterator, class T>
typename std::iterator_traits<
  InputIterator>::difference_type
count (InputIterator first,
       InputIterator last,
       const T& val)
{
  return count1(first,last,val);
}
```

The count algorithm is implementable with the accumulate algorithm and a custom lambda. The accumulate algorithm is described in the following section.

recursive::count

```
#include <numeric>

template <class InputIterator,
          class T>
typename std::iterator_traits<
  InputIterator>::difference_type
count (InputIterator first,
       InputIterator last,
       const T& val)
{
  return
    accumulate(
      first,
      last,

      0,

      [&](int x,int y){ return x+(y==val); });

}
```

accumulate

The accumulate algorithm is in the numeric header and it applies a binary operator to the “init” value passed into the function and “first” element of the sequence. The results of that operator are passed as the “init” value for the rest of the sequence. One of the most common uses of the accumulate algorithm is to add up the elements of a sequence and return the mathematical sum.

recursive::accumulate

```
template <class InputIterator,
          class T,
          class BinaryOperation>
T accumulate (InputIterator first,
              InputIterator last,
              T init,
              BinaryOperation op)
{
  if (first == last) return init;
  return
    accumulate(std::next(first),
               last,
               op(init,*first),
               op);
}
```

count_if

The count_if algorithm counts the number of times a predicate is true when applied to an element in a sequence. The algorithm uses the iterator_traits difference_type to determine the type of the result value.

recursive::count_if

```
template <class InputIterator,
          class UnaryPredicate>
typename std::iterator_traits<
  InputIterator>::difference_type
count_if (InputIterator first,
       InputIterator last,
       UnaryPredicate pred)

{

  if (first==last) return 0;

  return

    pred(*first) +

    count_if(std::next(first), last, pred);

}
```

mismatch

The mismatch algorithm finds the first elements in two sequences that do not match and returns a pair of iterators that point to the mismatching items. If all of the elements match, the iterator of the first sequence equals the last iterator.

recursive::mismatch

```
template <class InputIterator1,
          class InputIterator2>
std::pair<InputIterator1,InputIterator2>
mismatch (InputIterator1 first1,
          InputIterator1 last1,
          InputIterator2 first2)

{

  if (first1==last1)

    return std::make_pair(first1,first2);

  if (*first1!=*first2)

    return std::make_pair(first1,first2);

  return

    mismatch(std::next(first1),

             last1,

             std::next(first2));

}
```

It is important that the second sequence is not shorter than the first sequence or it may result in undefined behavior.

equal

The equal algorithm compares two sequences and returns true if the elements of the two sequences are equal. The algorithm is simply, implementable with the AND operator (&&).

recursive::equal

```
template <class InputIterator1,
          class InputIterator2>
bool equal(InputIterator1 first1,
           InputIterator1 last1,
           InputIterator2 first2)

{

  if (first1==last1) return true;

  return

    (*first1 == *first2) &&

    equal(std::next(first1),

          last1,

          std::next(first2));

}
```

advance

The advance algorithm advances an iterator “n” times, and the following recursive version returns the resulting iterator. Unlike the STL version it does not update the passed in iterator.

recursive::advance

```
template<class InputIterator,
         class Distance>
InputIterator
advance(InputIterator it, Distance n)
{
  return
    !n ? it : advance(std::next(it),n-1);
}
```

This simple function is used in the following is_permutation algorithm.

distance

The distance algorithm counts the number of items between the two iterators passed to the function.

recursive::distance

```
template<class InputIterator>
typename
std::iterator_traits<
  InputIterator>::difference_type
distance(InputIterator first
         InputIterator last)
{
  if (first==last) return 0;
  return 1+distance(std::next(first),last);
}
```

This simple function is also used in the following is_permutation algorithm.

is_permutation

The is_permutation algorithm determines if two sequences are permutations of each other. The two sequences contain equal elements in possibly different orders. The algorithm uses mismatch to start the algorithm after the beginning subsequences that already match. After that it calls the helper is_permutation1 function, which iterates over the elements of the first sequence. It checks if that element has been encountered before using the find algorithm, and if not it compares the number of occurrences in both sequences to see if they match. The count comparison is logically ANDed with the recursive call to is_permutation1.

recursive::is_permutation

```
template <class InputIterator1,
          class InputIterator2>
bool
is_permutation1 (InputIterator1 first1,
                 InputIterator1 curr,
                 InputIterator1 last1,
                 InputIterator2 first2,
                 InputIterator2 last2)
{
  if (curr==last1) return true;
  if (find(first1,curr,*curr)==curr)
    return
     ((1+count(std::next(curr),last1,*curr))
     == count(first2,last2,*curr)) &&
       is_permutation1(first1,
                       std::next(curr),
                       last1,
                       first2,
                       last2);
  else return
    is_permutation1(
      first1,
      std::next(curr),
      last1,
      first2,
      last2);
}

template <class InputIterator1,
          class InputIterator2>
bool
is_permutation(InputIterator1 first1,
               InputIterator1 last1,
               InputIterator2 first2)

{
  auto start = mismatch(first1,last1,first2);
  if (start.first==last1) return true;
  return
    is_permutation1(
      start.first,
      start.first,
      last1,
      start.second,
      advance(
        start.second,
        distance(start.first,last1)));
}
```

An optimized implementation is possible that checks if the count of the elements in the second sequence is greater than 1 before counting the elements in the first sequence. This is left to the reader as an exercise.

search

The search algorithm finds the first occurrence of a subsequence in a sequence. It first checks if the subsequence has zero elements and if so returns the first iterator of the sequence. Otherwise, it calls the search1 helper function, which iterates over the sequence and checks each element as the start of the subsequence. The subsequence is checks using the starts with function used previously with the find_end algorithm.

recursive::search

```
template<class ForwardIterator1,
         class ForwardIterator2>
bool
starts_with(
  ForwardIterator1 first1,
  ForwardIterator1 last1,
  ForwardIterator2 first2,
  ForwardIterator2 last2)
{
  if (first2==last2) return true;
  if (first1==last1) return false;
  return (*first1 == *first2) &&
    starts_with(std::next(first1),
                last1,
                std::next(first2),
                last2);
}

template<class ForwardIterator1,
         class ForwardIterator2>
ForwardIterator1
search1(ForwardIterator1 first1,
        ForwardIterator1 last1,
        ForwardIterator2 first2,
        ForwardIterator2 last2
)
{
```

```cpp
  if (first1==last1) return last1;

  if (starts_with(first1,last1,first2,last2))

    return first1;

  return

    search1(std::next(first1),

            last1,

            first2,

            last2);

}


template<class ForwardIterator1,
         class ForwardIterator2>
ForwardIterator1
search (ForwardIterator1 first1,
        ForwardIterator1 last1,
        ForwardIterator2 first2,
        ForwardIterator2 last2)

{

  if (first2==last2) return first1;

  return

    search1(first1,last1,first2,last2);

}
```

search_n

The search_n algorithm searches a sequence for an item that equals the passed value and which is repeated “n” times. The returned result is an iterator to the first instance of the repeated sequence. The search_n function calls the helper function search_n1 which recurses over a sequence and reduces the count by one when the item matches the passed value. It returns true when the count reaches zero, or returns false when the match fails. When it returns false it also returns the next location where the search can restart to look for the string of matches, so the search will not iterate over elements more than once. The iterator returned with the true value is meaningless and not used.

recursive::search_n

```
template<class ForwardIterator,
         class Size,

         class T>

std::pair<bool,ForwardIterator>

search_n1(ForwardIterator first,

          ForwardIterator last,

          Size count,

          const T& val)

{

  if (!count)

    return std::make_pair(true,first);

  if (first==last)

    return std::make_pair(false,last);

  if (*first != val)

    return std::make_pair(

             false,

             std::next(first));

  return search_n1(

           std::next(first),

           last,

           count-1,

           val);
}


template<class ForwardIterator,
         class Size,
         class T>
ForwardIterator
search_n (ForwardIterator first,
          ForwardIterator last,
          Size count,
          const T& val)

{

  const auto pair =

    search_n1(first,last,count,val);

  if (pair.first) return first;

  if (pair.second==last) return last;

  return

    search_n(

      pair.second,

      last,

      count,

      val);

}
```

min_element

The min_element algorithm returns an iterator to the minimum element in a sequence of items. The minimum value is retrieved by dereferencing the returned iterator. The algorithm uses a helper function to pass the minimum element as the algorithm recurses.

recursive::min_element

```
template <class InputIterator,
          class T>
InputIterator
min_element1(InputIterator first,
             InputIterator last,
             InputIterator min)
{
  if (first == last) return min;
  return
    min_element1(
      std::next(first),
      last,
      (*min < *first) ? min : first);
}

template <class InputIterator>
InputIterator
min_element (InputIterator first,
             InputIterator last)
{
  if (first == last) return last;
  return
    min_element1(
      std::next(first),
      last,
      first);
}
```

An implementation of the max_element or minmax_element would similarly pass the iterator to the maximum element as the algorithm recurses. An implementation of these algorithms is left as an exercise to the reader.

adjacent_difference

The adjacent_difference algorithm in the numeric header assigns the adjacent difference of each element and the previous element of a sequence is assigned to the elements on a result sequence. This algorithm is not purely functional, in that it does assign the result to an output iterator. It was implemented recursively, but not without assignment. The algorithm uses a helper function adjacent_difference1 that recurses over the elements after the first element.

recursive::adjacent_difference

```
template<class InputIterator,
         class OutputIterator>
OutputIterator
adjacent_difference1(
  InputIterator first,
  InputIterator next,
  InputIterator last,
  OutputIterator result)
{
  if (next==last) return result;
  *result = *next - *first;
  return adjacent_difference1(
    next,std::next(next),last,std::next(result));
}

template<class InputIterator,
         class OutputIterator>
OutputIterator
adjacent_difference(
  InputIterator first,
  InputIterator last,
  OutputIterator result)
{
  if (first==last) return result;
  *result = *first;
  return adjacent_difference1(
    first,std::next(first),last,
    std::next(result));
}
```

inner_product

The inner_product algorithm in the numeric header accumulates the inner_product of the elements of two sequences.

recursive::inner_product

```
template <class InputIterator1,
          class InputIterator2,
          class T>
T
inner_product(
  InputIterator1 first1,
  InputIterator1 last1,
  InputIterator2 first2,
  T init)
{
  if (first == last) return init;
  return
    inner_product(
      std::next(first1),
      last1,
      std::next(first2),
      init+(*first1)*(*first2));
}
```

partial_sum

The partial_sum algorithm in the numeric header calculates the partial sum of the elements in a sequence. The “nth” element in the output sequence is assigned the sum of the first “n” elements in the input sequence. The partial_sum algorithm uses assignment to set the output values, so it is not strictly free of side effects. This implementation maintains the typical STL interface with a recursive definition that uses the partial_sum1 helper function.

recursive::partial_sum

```
template <class InputIterator,
          class OutputIterator>
OutputIterator
partial_sum1(
  InputIterator first,
  InputIterator last,
  OutputIterator result,
  typename std::iterator_traits<
    OutputIterator>::value_type val)
{
  if (first==last) return result;
  return
  partial_sum1(
    std::next(first),
    last,
    std::next(result),
    *result = *first + val);
}

template <class InputIterator,
          class OutputIterator>
OutputIterator
```

```
partial_sum(
  InputIterator first,
  InputIterator last,
  OutputIterator result)
{
  return
    partial_sum1(first,last,result,0);
}
```

A different version of the partial_sum algorithm that returns the output data sequence instead of assigning the data to an output iterator encapsulates the algorithm in a functional interface that calls partial_sum1 and the previously defined distance function. A vector is constructed in the function with a size of the input sequence, and the begin iterator of the sequence is passed as the output iterator.

recursive::partial_sum

```
template <class InputIterator,
          class OutputIterator>
std::vector<
  typename std::iterator_traits<
    InputIterator>::value_type>
partial_sum(
  InputIterator first,
  InputIterator last)
{
  std::vector<
    typename std::iterator_traits<
      InputIterator>::value_type>
    v(distance(first,last));
  return
    partial_sum1(
      first,
        last,
        std::begin(v),
        0);
}
```

iota

The iota algorithm assigns an iterator an incrementing sequence of values starting from the input value. The algorithm assigns the values to an input iterator, so the function has a side effect. However, it is implementable recursively with the standard STL interface as follows.

recursive::iota

```
template <class ForwardIterator,
          class T>
void
iota(
  ForwardIterator first,
  ForwardIterator last,
  T val)
{
  if (first == last) return;
  *first = val;
  iota(std::next(first),last,val+1);
}
```

A version of this algorithm that is functional would need to pass in the number of elements to create which is implicitly found in the distance between the first and last iterator of the previous implementation. This value could be retrieved by calling the begin and end iterator with the STL distance function. The following implementation calls the previous version and returns a vector with the increasing elements.

recursive::iota

```
template <class T,
          class Size>
```

```
std::vector<T>
iota(T val, Size n)
{
  std::vector<T> v(n);
  iota(std::begin(v),std::end(v),val);
  return v;
}
```

Run-time Results

Each recursive algorithm was run vs. the STL implementation on a long sequence in a vector, list, and set. The following chart shows which algorithm was faster when run with a large number of iterations.

Algorithm	**vector**	**list**	**set**
all_of	recursive	same	recursive
any_of	same	same	recursive
none_of	recursive	same	same
for_each	recursive	recursive	recursive
find	recursive	recursive	recursive
find_if	same	STL	recursive
find_if_not	STL	same	STL
find_end	same	STL	recursive
find_first_of	same	STL	recursive
adjacent_find	STL	recursive	N/A
count	STL	same	recursive
count_if	STL	same	recursive
accumulate	STL	recursive	STL

mismatch	STL	recursive	STL
equal	STL	STL	recursive
is_permutation	STL	recursive	N/A
search	STL	STL	STL
search_n	STL	recursive	N/A
adjacent_difference	STL	recursive	STL
inner_product	STL	recursive	recursive

The difference in speed when one implementation won was often negligible. The point of this analysis is not that one implementation should be used over the other based on speed, but that a recursive implementation is not necessarily a detriment to run-time performance.

constexpr

The constexpr keyword in C++11 allows defining functions that are evaluated at runtime. All inputs to a constexpr function must also be constexpr. In C++11 these functions should be single line functions and can be recursive. Many of the previous functions support this when the ternary operator is used to handle the recursion base case. See the advance algorithm in this chapter for an example. An array sequence could be used with that algorithm since its iterators can be returned with constexpr begin/end function. The following sum function is executed at compile time, thus resulting in only the printing with cout at runtime.

main.cpp (C++11 and C++14)

```
#include <iostream>
#include <array>

template<typename T>
constexpr int sum(T b, T e)
{
  return (b==e) ? 0 : *b + sum(std::next(b),e);
}

int main()
{
  std::array<int,3> a = { 1,2,3 };
  std::cout << sum(a.begin(),a.end()) << std::endl;
  return 0;
}
```

C++14 allows constexpr functions with conditionals and loops, and it supports all of the previous algorithms. None of the previous algorithms were defined as constexpr for simplicity. The following sum function with a for loop is executed at compile time.

main.cpp (C++14 only)

```
#include <iostream>
#include <array>

template<typename T>
constexpr int sum(T b, T e)
{
  int result = 0;
  for(;b!=e;++b)
  {
    result += *b;
  }
  return result;
}

int main()
{
  std::array<int,3> a = { 1,2,3 };
  std::cout << sum(a.begin(),a.end()) << std::endl;
  return 0;
}
```

Lambdas

C++11 introduced lambda functions, or lambdas, as a language construct which supports defining a function where the function is used. The function is unnamed where defined and referred to as an anonymous function. However, a lambda can be assigned to a named variable for convenience. A variable that holds a lambda must be declared with the type “auto” or a lambda can be assigned to a STL function object, std::function. Lambdas that are passed to a function as a function object, must use template arguments for the compiler to deduce the type of the lambda function.

The typical lambda consists of the capture clause, [], parameter list, (), return type, -> T, and lambda function body, {}. These are all typical of a regular function with the new, arrow notation for the return type, ->, and the added capture clause to describe how variables are captured for use in the lambda body. The following example rounding lambda, rounds a floating point number to the nearest integer:

lambda

```
[](float x) -> int
{
    return int(x+0.5f)
}
```

The following code uses the functor to transform a vector of floating point values to integer values. The return type is not necessary in this case and left undefined.

main.cpp

```
int main(void)
{
  std::vector<float> v
    = { 0,1.1f,2.2f,3.3f,4.4f,5.5f,6.6f };
  std::vector<int> y(v.size());
  func_wrapper<int(int)> fib;
  std::transform(
    std::begin(v),
    std::end(v),
    std::begin(y),
    [](float x)
    {
    return int(x+0.5f);
    });

  for(int i=0; i<y.size(); ++i)
    std::cout << y[i] << std::endl;
  return 0;
}
```

Recursive Lambda

Recursive functions often represent a succinct and efficient implementation for many algorithms. Therefore, a recursive lambda is an interesting twist on lambdas. The lambda must first be declared with a name so that the name is referenced recursively in itself. The following code shows a Fibonacci function, fib, that is declared and recursively calls itself in the lambda passed to the transform algorithm.

main.cpp

```
int main(void)
{
  std::vector<int> v = { 0,1,2,3,4,5 };
  std::vector<int> y(v.size());
  func_wrapper<int(int)> fib;
  std::transform(
    std::begin(v),
    std::end(v),
    std::begin(y),
    fib = [fib](int n)
    {
    return (n <= 2)? 1 : fib(n-1) + fib(n-2);
    });

  for(int i=0; i<y.size(); ++i)
    std::cout << y[i] << std::endl;
  return 0;
}
```

The recursive calls to fib are accomplished with the following func_wrapper that maintains a shared pointer to the function object.

recursive::lambda

```
template <class T>
class func_wrapper
{
  std::shared_ptr<std::function<T>> func;
  func_wrapper *captured;

public:

  func_wrapper()
   : captured(nullptr)
  {}

  func_wrapper(const func_wrapper &) = default;
  func_wrapper(func_wrapper &&) = default;

  // non-const copy-ctor
  func_wrapper(func_wrapper & f)
   : func(f.func),
   captured(nullptr)
  {
   f.captured = this;
  }

  template <class U>
  const func_wrapper & operator = (U&& closure)
  {
   func = std::make_shared<std::function<T>>();
   if (captured)
   captured->func = func;

   (*func) = std::forward<U>(closure);
   return *this;
  }

  template <class... Args>
  auto operator () (Args&&... args) const
   -> decltype((*func)(std::forward<Args>(args)…))
  {
   return (*func)(std::forward<Args>(args)…);
  }
};
```

Memoization

A pure function (without side effects) always returns the same result when called with the same arguments. Storing the result of the first call to a function for later retrieval, results in the ability to get the cached value with the same set of arguments passed in previously. For functions that take a long time to run, this caching and retrieval can result in a run-time savings. This technique in functional programming is called memoization. A generic memoizer that stores and retrieves the arguments and result from a function object is implementable with a hash_map. The following implementation uses the hash function found in the Appendix to store a function objects arguments and results the first time a set of arguments are used. After that it will find the result generated from these arguments in the hash map named cache. This cache is set to be mutable for using the function where it expects the function to be const.

memoizer.hpp

```
template<class Sig, class F,
    template<class…> class Hash=std::hash>
struct memoizer;


template<class R, class…Args,
    class F, template<class…> class Hash>
struct memoizer<R(Args…), F, Hash> {
  using base_type = F;
private:
  F base;
  mutable std::unordered_map<
    std::tuple<std::decay_t<Args>…>,
    R,
    Hash<std::tuple<std::decay_t<Args>…>>> cache;


public:


  template<class… Ts>
```

```
R operator()(Ts&&… ts) const
{
  auto args = std::make_tuple(ts…);
  auto it = cache.find( args );
  if (it != cache.end())
  return it->second;

  auto&& retval = base(std::forward<Ts>(ts)…);

  cache.emplace( std::move(args), retval );

  return decltype(retval)(retval);
}
template<class… Ts>
R operator()(Ts&&… ts)
{
  auto args = std::tie(ts…);
  auto it = cache.find( args );
  if (it != cache.end())
  return it->second;

  auto&& retval = base(std::forward<Ts>(ts)…);

  cache.emplace( std::move(args), retval );

  return decltype(retval)(retval);
}

memoizer(memoizer const&)=default;
memoizer(memoizer&&)=default;
memoizer& operator=(memoizer const&)=default;
memoizer& operator=(memoizer&&)=default;
memoizer() = delete;
```

```
  template<typename L>
  memoizer( wrap, L&& f ):
    base( std::forward<L>(f) )
  {}
};

template<class Sig, class F>
memoizer<Sig, std::decay_t<F>> memoize( F&& f )
{ return {wrap{}, std::forward<F>(f)}; }
```

The following main function shows how to use the memoizer with a function object (sqrt_double_num).

main.cpp

```
class sqrt_double_num
{
public:
  double operator()(double x) const
  {
    // Long calculation
    for(int i=0;i<100000000;++i)
    x = sqrt(x*x);
    return x;
  }
};

int main()
{
  const auto f
    = memoize<
    double(double)>(sqrt_double_num());
  std::cout << f(2.) << std::endl;
  std::cout << f(2.) << std::endl;
  return 0;
}
```

The memoize function creates the memoizer object with the sqrt_double_num function object. This function caches the result from the first call to “f”, and the second call looks up the result in the cache. The memoizer does not support recursion, but a small change to the memoizer and the function object will fix that.

recursive memoizer

Change

```
auto&& retval = base(std::forward<Ts>(ts)…);
```

To

```
auto&& retval = base(*this,std::forward<Ts>(ts)…);
```

Passing the memoizer to the function object allows the function object to make the recursive call to the memoizer. The signature of the parens operator requires adding the memoizer object as the first argument to the function object. The memoizer is called to retrieve the result of a call to the function object with a different argument as follows.

main.cpp

```
class factorial
{
public:
  double operator()(
    const memoizer<double(double),factorial>& m,
    double x) const
  {
    // Long calculation
    if(x==1) return 1;
    return x*m(x-1);
  }
};


int main()
{
  const auto f
    = memoize<
    double(double)>(factorial());
  std::cout << f(10.) << std::endl;
  std::cout << f(10.) << std::endl;
  return 0;
}
```

Lazy Evaluation

The typical evaluation mode of C++ is eager evaluation, such that a function that is passed arguments evaluates the function immediately. The following example evaluates the summation of both the even and odd vectors when the “eager” function is called.

main.cpp

```
#include <iostream>

#include <numeric>

#include <vector>


enum numbers { EVEN, ODD };


template<typename T>

int eager(numbers n, T even, T odd)

{

  if(n == EVEN) return even;

  return odd;

}


int add(const std::vector<int>& v)

{

  return std::accumulate(v.begin(),v.end(),0);

}


int main()

{

  std::vector<int> even = { 0,2,4,6,8 };

  std::vector<int> odd = { 1,3,5,7,9 };

  numbers n = EVEN;

  std::cout << eager(n,add(even),add(odd));
  return 0;
}
```

While the result of only the even summation is used, the compiler will generate code that also computes the summation of the odd numbers.

Alternatively, C++11 includes the async function that is passed a function and arguments to pass to the function, but instead of immediately evaluating the function it supports delaying evaluation until the result value is accessed. The async function returns a future from which the result value is accessed. The function is run in the current thread of the code accessing the variable.

main.cpp

```
#include <future>
#include <iostream>
#include <numeric>
#include <vector>

enum numbers { EVEN, ODD };

template<typename T>
int lazy(numbers n, T even, T odd)
{
  if(n == EVEN) return even.get();
  return odd.get();
}

int add(const std::vector<int>& v)
{
  return std::accumulate(v.begin(),v.end(),0);
}
```

```
int main()
{
  std::vector<int> even = { 0,2,4,6,8 };
  std::vector<int> odd = { 1,3,5,7,9 };
  numbers n = EVEN;
  std::cout <<
    lazy(EVEN,
    std::async(std::launch::deferred,add,even),
    std::async(std::launch::deferred,add,odd));
  return 0;
}
```

The result of evaluating the function passed to async, is accessed from the future with the “get” function call in the “lazy” function.

When the value of a future is accessed a second or more time, the value is not recalculated and returned using memoization. Memoization was discussed in the previous chapter.

The previous example does not use memorization and could be achieved with a standard function object as shown in the following example. The object is constructed with the function’s argument, and the function is evaluated when the parentheses operator is called.

main.cpp

```
#include <future>
#include <iostream>
#include <numeric>
#include <vector>

enum numbers { EVEN, ODD };
```

```
template<typename T,typename U>
int lazy(numbers n, T even, U odd)
{
  if(n == EVEN) return even();
  return odd();
}

class functor
{
public:
  functor(const std::vector<int>& v) : v_(v) {}

  int operator()() const
  {
    return std::accumulate(v_.begin(),v_.end(),0);
  }
private:
  const std::vector<int>& v_;
};

int add(const std::vector<int>& v)
{
  return std::accumulate(v.begin(),v.end(),0);
}

int main()
{
  std::vector<int> even = { 0,2,4,6,8 };
  std::vector<int> odd = { 1,3,5,7,9 };
  auto even_future
    = std::async(std::launch::deferred,add,even);
  auto odd_future
    = std::async(std::launch::deferred,add,odd);
  numbers n = EVEN;
  std::cout
    << lazy(n,functor(even),functor(odd))
    << std::endl;
  std::cout <<
    lazy(n,
    [&](){ return even_future.get(); },
       [&](){ return odd_future.get(); })
    << std::endl;
}
```

The `lazy` function calls the parentheses operator of the functor to access the value when it’s needed. Writing a functor like this requires a lot of boiler-plate code, and this function is also called similarly on the following line with a future from the async function by wrapping the get function with a lambda.

Templates

While many of the features C++ inherited from C, such as const, functions, recursion, etc. are typical of functional and imperative programming, C++'s template system has long been acknowledged as purely functional. It is a functional language in it's own right that is Turing complete. Templates generally calculate through C++'s type system, but also can calculate with non-types, such as int or enums. Programming using C++ templates has spawned the field of "Template Metaprogramming" that consists of compile-time calculations. Many C++ compilers include partial evaluation to optimize a program at compile time by evaluating operations, however, there is no guarantee that these optimizations will be done. Template Metaprogramming operations must be performed due to the C++ standard requiring that template arguments and types must be fully specified at compile time.

main.cpp

```
#include <iostream>


template<int n>

struct factorial

{

  enum { result = n * factorial<n-1>::result };

};


template<>

struct factorial<0>

{

  enum { result = 1 };

};


int main()

{

  std::cout << "factorial<10>::result = " << factorial<10>::result << std::endl;

  return 0;
}
```

A template struct acts as a function with the template arguments as inputs, the enum variable as the output, and the calculation as the right-hand side of the enum initialization. The calculation in the factorial struct is recursive and terminates at zero using a template specialization.

A template function can also use conditional statements and have multiple results as shown in the following example comparing two integers with the ternary operator.

main.cpp

```
#include <iostream>

template<int A, int B>
struct compare
{
  enum { min = A < B ? A : B };
  enum { max = A > B ? A : B };
};

int main()
{
  std::cout << “compare<1,100>::min = “
    << compare<1,100>::min << std::endl;
  std::cout << “compare<1,100>::max = “
     << compare<1,100>::max << std::endl;
  return 0;
}
```

C++11 has expanded compile-time calculation using the constexpr keyword that includes floating-point variables, such as the following:

main.cpp

```
#include <iostream>


int main()
{
  constexpr double A = 3.14159;
  constexpr double B = 100.0;
  constexpr double max = A > B ? A : B;
  constexpr double min = A < B ? A : B;

  std::cout << “max(A,B) = ” << max << std::endl;
  std::cout << “min(A,B) = ” << min << std::endl;
  return 0;
}
```

A constexpr calculation must have all input variables also be constexpr. It would seem that an input variable to a constexpr could be just const, but I found that the clang compiler wouldn’t accept that. Function can also be declared as constexpr for compile-time calculation as follows:

main.cpp

```
#include <iostream>


constexpr double min(double A, double B)
{
  return A < B ? A : B;
}
```

```
constexpr double max(double A, double B)

{

  return A > B ? A : B;

}


int main()

{

  constexpr double A = 3.14159;

  constexpr double B = 100;


  std::cout << “A > B = ” << max(A,B) << std::endl;

  std::cout << “A < B = ” << min(A,B) << std::endl;

  return 0;

}
```

Common functional data structures, such as lists, can also be manipulated using template metaprogramming. A list uses a nil termination node and element nodes that hold a value and a link to the next node. These are implemented with a simple nil struct and an element node with two template arguments. The template arguments represent the element value (head) and the link (tail). A struct/class does not expose its template arguments outside of its scope, so an enum exposes the value and the typedef exposes the tail.

main.cpp

```
#include <iostream>


struct nil {};


template<int head_, typename tail_>

struct node

{

  enum { head = head_ };

  typedef tail_ tail;
};


int main()

{

  typedef node<3, node<2, node<1, node<0, nil>>>> list;

  return 0;

}
```

The template-based list is captured in the type definition of list without an instance being instantiated. The “at” algorithm, shown below, extracts the values of the list using a recursive call to itself and a partial template specialization for the tail call. (The nil and node structs were put in “list.h”.)

at.cpp

```
#include <iostream>

#include “list.h”


template<typename T,int N>

struct at

{

  enum { value = at<typename T::tail,N-1>::value };

};


template<typename T>

struct at<T,0>

{

  enum { value = T::head };

};


int main()

{
  typedef node<3, node<2, node<1, node<0, nil>>>> list;


  std::cout << at<list,0>::value << std::endl;

  std::cout << at<list,1>::value << std::endl;

  std::cout << at<list,2>::value << std::endl;

  std::cout << at<list,3>::value << std::endl;

  return 0;

}
```

The result of the “at” algorithm is captured in the “value” enum, which also exposes it outside of the “at” struct, similar to the “head” enum of the list node. A similar algorithm that results in the size of the list is shown below.

size.cpp

```
#include <iostream>

#include “list.h”


template<typename T>

struct size

{

  enum { value = 1 + size<typename T::tail>::value };

};


template<>

struct size<nil>

{

  enum { value = 0 };

};


int main()

{
  typedef node<3, node<2, node<1, node<0, nil>>>> list;


  std::cout << size<list>::value << std::endl;

  return 0;

}
```

Typical functions like “if” are easily implementable as metaprogramming functions.

if.cpp

```
#include <iostream>


template<bool X, typename T, typename F>

struct IF

{};


template<typename T, typename F>

struct IF<true,T,F>

{

  typedef T type;

};


template<typename T, typename F>

struct IF<false,T,F>

{

  typedef F type;

};


int main()

{

  typename IF<true,int,float>::type vi = 1.1f;
  typename IF<false,int,float>::type vf = 1.1f;


  std::cout << vi << std::endl;

  std::cout << vf << std::endl;

  return 0;

}
```

Creating template data structures and algorithms is important to understand how template metaprogramming works, however, it’s smarter to not reinvent the wheel. Two very powerful, template metaprogramming libraries, Boost MPL [2] and Boost Fusion [3] already include many, useful data structures, algorithms, and functions.

Boost MPL

The Boost Metaprogramming Library (MPL) is a thorough, template metaprogramming library for performing compile-time type manipulation, including data structures (compile-time vector, list, set, map), iterators, views, metafunctions, algorithms, and traits. This library implemented most of the standard C++ abstractions in compile-time versions. It also supports lazy evaluation of metafunctions such that functions can be formed which would result in uncompilable types, but they will not result in compile errors if they are not evaluated. The Boost MPL documentation is available online, and a book by the authors, “C++ Template Metaprogramming” [4] includes an excellent explanation and many examples.

The following code shows a bit of what is possible with Boost MPL. A collection of types are transformed to the commensurate pointer types for each element using the add_pointer metafunction. The last line, BOOST_MPL_ASSERT, is a compile time check that the transform yields the same types as in vec_ptr. The “type” in the transform algorithm yields the result of the add_pointer metafunction. The metafunction uses the underscore placeholder “_” to represent that the metafunction is applied to the element it is passed.

main.cpp

```
#include <boost/mpl/vector.hpp>
#include <boost/mpl/transform.hpp>
#include <boost/mpl/equal.hpp>
#include <string>

int main()
{
  typedef boost::mpl::vector<int, float, std::string > vec;
  typedef boost::mpl::vector<int*,float*,std::string*> vec_ptr;
  typedef boost::mpl::_ _;

  typedef std::add_pointer<_> fx;

  typedef boost::mpl::transform<vec,fx>::type result;
  BOOST_MPL_ASSERT(( boost::mpl::equal<result,vec_ptr> ));
}
```

This barely scratches the surface of what is possible with Boost MPL.

Boost Fusion

Boost Fusion expands on the MPL by straddling the compile-time and run-time divide with data structures and algorithms that manipulate compile-time structure with run-time values. A typical example is a fixed size array or tuple, but Boost Fusion expands on these with lists, maps, sets, MPL structures, and more. The library also includes iterators, views, and algorithms, and it's extensible to support custom data structures.

Tuples are a convenient way to process a set of variables in a consistent way. The following code prints the elements of a tuple of different types, regardless of whether they type is an integral, floating-point, or string type. The fusion, for_each algorithm applies the print metafunction to each element of the tuple.

main.cpp

```
#include <boost/fusion/tuple.hpp>

#include <boost/fusion/algorithm.hpp>

#include <iostream>

#include <string>


struct print

{

  template <typename T>

  void operator()(const T &t) const

  {

    std::cout << t << std::endl;

  }

};


int main()

{

  typedef boost::fusion::tuple<int,float,std::string> tuple_type;

  tuple_type t(2, 3.3, “Chris”);

  boost::fusion::for_each(t,print());
}
```

Persistent Data

With most computers including multiple cores the fastest way to speed up a program is to use threading. Persistent data structures that don’t change their values are automatically thread-safe, while, most, common (non-persistent) C++ data structures suffer from the need to use mutexes to carefully manage adding, updating, and removing values. The following singly-linked list creates a new list when items are added to the list, while reusing the items at the tail of the list. [5]

main.cpp

```
#include <cassert>

#include <functional>

#include <iostream>

#include <initializer_list>


template<class T>

class List

{

  struct Item

  {

   Item(T v, std::shared_ptr<const Item> const & tail)

   : _val(v), _next(tail) {}

   T _val;

   std::shared_ptr<const Item> _next;

  };

  friend Item;

  explicit List (std::shared_ptr<const Item> const & items) :
_head(items) {}

public:

  // Empty list
```

```
List() : _head() {}
// Cons
List(T v, List tail)
 : _head(std::make_shared<Item>(v, tail._head)) {}
// From initializer list
List(std::initializer_list<T> init) : _head()
{
 for (auto it = std::rbegin(init);
 it != std::rend(init);
 ++it)
 {
 _head = std::make_shared(*it, _head);
 }
}

bool isEmpty() const { return !_head; }
T front() const
{
 assert(!isEmpty());
 return _head->_val;
}
List popped_front() const
{
 assert(!isEmpty());
 return List(_head->_next);
}
// Additional utilities
List pushed_front(T v) const
{
   return List(v, *this);
  }
  List insertedAt(int i, T v) const
  {
   if (i == 0)
   return pushed_front(v);
   else {
   assert(!isEmpty());
   return
   List(front(),
   popped_front().insertedAt(i - 1, v));
   }
  }
private:
  // Encode a Maybe value (nullptr is empty list)
  std::shared_ptr<const Item> _head;
};
```

List Functions

The first function recursively generates a List from a pair of iterators, and works well for processing the data in a standard container.

fromIterator.hpp

```
template<class Beg, class End>
auto
fromIterator(Beg it, End end)
  -> List<typename Beg::value_type>
{
  typedef typename Beg::value_type T;
  if (it == end) return List<T>();
  return List<T>(*it, fromIt(std::next(it), end));
}
```

This typical list data structure of languages like LISP can be processed by many canonical LISP functions. Anyone who has studied LISP will recognize these functions. The concat function concatenates two lists as follows:

concat.hpp

```
template<class T>
List<T>
concat(List<T> a, List<T> b)
{
  if (a.isEmpty()) return b;
  return List<T>(a.front(), concat(a.popped_front(), b));
}
```

The following functional, list functions are useful for processing the elements of a list. fmap returns a list with the elements of applying the input function to each element of a list. fmap applies f to the first element and recursively calls the tail with fmap.

fmap.hpp

```
template<class U, class T, class F>
List<U>
fmap(F f, List<T> lst)
{
  if (lst.isEmpty()) return List<U>();
  else
   return List<U>(f(lst.front()),
         fmap<U>(f, lst.popped_front()));
}
```

The following, filter function applies a functor to each element of a list and returns a list with the elements where the functor returns true.

filter.hpp

```
template<class T, class P>
List<T>
filter(P p, List<T> lst)
{
  if (lst.isEmpty()) return List<T>();
  if (p(lst.front()))
   return List<T>(
   lst.front(),
   filter(p, lst.popped_front()));
  else
   return filter(p, lst.popped_front());
}
```

The STL function accumulate is a functional “fold” operation that accumulates the values in a list. In functional programming there is a “right” and “left” fold dependent upon which direction the items accumulate from.

foldr.hpp

```
template<class T, class U, class F>
U
foldr(F f, U acc, List<T> lst)
{
  if (lst.isEmpty()) return acc;
  else
   return f(lst.front(),
   foldr(f, acc, lst.popped_front()));
}
```

A commutative functor would return the same result for the previous “right” (foldr) fold or the following “left” (foldl) fold.

foldl.hpp

```
template<class T, class U, class F>
U
foldl(F f, U acc, List<T> lst)
{
  if (lst.isEmpty()) return acc;
  else
   return foldl(f,
   f(acc,lst.front()),
   lst.popped_front());
}
```

The following forEach function applies a functor to each element of a List.

forEach.hpp

```
template<class T, class F>
void forEach(List<T> lst, F f)
{
  if (!lst.isEmpty()) {
   f(lst.front());
   forEach(lst.popped_front(), f);
  }
}
```

Finally for debugging the following print function, which uses the forEach function, will print the elements of a List.

print.hpp

```
template<class T>

void

print(List<T> lst)

{

  forEach(lst, [](const T& v)

  {

   std::cout << “(” << v << “) “;

  });

  std::cout << std::endl;

}
```

This list data structure suffers from poor data locality due to all of the elements being held by a shared pointer. Adding elements to a list is quick, but data accesses would suffer from poor cache performance. A persistent vector that creates a new vector with each change operation has good access, cache performance, but requires at least one large data copy with each change.

Persistent Vector

While a linked list data structure is optimal for adding items to a persistent data structure, it offers poor performance for sequential access. A vector excels in that case due to data elements in contiguous memory arrays that the CPU will cache for optimal performance. A vector is even more efficient for random access versus a linked list since the element offset from the array beginning is easily computed compared to traversing a list to find a particular element. A vector has O(1) complexity for random access versus O(n) for a linked list.

A persistent vector does not allow setting elements or appending elements without copying the data to create a new vector. The following vector implementation is stripped down to constructors, a getter, a setter, push_back, size, and begin/end iterators. This covers the most common functions of the STL vector and implements them such that they only work on a const vector.

vector.hpp

```
#include <vector>


template<typename T>

class vector

{

public:

  typedef

    typename std::vector<T>::const_iterator

    const_iterator;


  vector() : v_() {}


  vector(std::size_t s,T init)

  : v_(s,init)

  {}


  vector(const vector<T>& v)
```

```
    : v_(v.v_)
  {}

  vector(vector<T>&& other)
    : v_(std::move(other.v_))
  {}

  std::size_t size() const { return v_.size(); }

  vector<T> set(std::size_t i, const T& t) const
  {
    vector<T> result(*this);
    result.v_[i] = t;
    return result;
  }

  T operator[](std::size_t i) const
  {
    return v_[i];
  }

  vector<T> push_back(const T& x) const
  {
    vector<T> result(*this);
    result.v_.push_back(x);
    return result;
  }

  const_iterator begin() const { return v_.begin(); }
  const_iterator end() const { return v_.end(); }
private:
  std::vector<T> v_;
};
```

The operator= function is left off since element and vector assignment is one operation that is forbidden in the form of a typical STL vector. Instead a set function is implemented to support setting a particular element. This operation is expensive since it requires copying the entire vector to set just one element.

All of the member functions are declared const so the persistent vector object needs to be constructed as a const vector. This example shows a vector, v, constructed with 5 elements initialized to zero, and a vector, v1, created from pushing a one on the back of v.

main.cpp

```
#include <iostream>
#include “vector.hpp”

int main()
{
  const vector<int> v(5,0);
  const auto v1 = v.push_back(1);
  for(size_t i = 0; i < v1.size(); ++i)
    std::cout << v1[i] << std::endl;
  return 0;
}
```

Since the vector is persistent, the push_back function must copy the vector before adding the additional element. The set function assigns a new value to an element of the vector, and also must copy the vector before updating the particular element. This vector copy in the push_back and set function is expensive, and should be used sparingly to since it is a O(n) operation. While this copy penalty may seem excessive it might not be fatal for programs that would only perform this copy during program initialization, and therefore it would not hurt the long-term running performance.

main.cpp

```
#include <iostream>

#include <numeric>

#include “vector.hpp”


int main()

{

  const vector<int> v(5,0);

  const auto v2 = v.set(1,9);

  for(size_t i = 0;i < v2.size();++i)

    std::cout << v2[i] << std::endl;


  return 0;

}
```

The const_iterators returned from the begin and end function can be used with standard STL algorithms like the accumulate function.

main.cpp

```
#include <iostream>

#include <numeric>

#include “vector.hpp”


int main()

{

  const vector<int> v(5,0);

  const auto v1 = v.set(1,9);

  std::cout

    << std::accumulate(v1.begin(),v1.end(),0)

    << std::endl;
  return 0;
}
```

The STL vector includes many more functions including assign, insert, erase, etc. These functions can be implemented for a persistent vector, and are left as an exercise for the reader.

More Persistent Data Structures

Additional STL data structures, such as map, unordered_map, list, etc., could be implemented as persistent similarly to the previous vector data structure. They would require copying data when the data structures are updated.

Appendix

Tuple Hash

To store and retrieve values from an unordered map with tuples for keys, the values need to be combined via hash function.

main.cpp

```
#include <tuple>

#include <functional>

#include <iostream>


size_t hash_combiner(size_t left, size_t right) //replaceable

{ return left^right;}


template<int index, class…types>

struct hash_impl {

size_t

operator()(size_t a,

    const std::tuple<types…>& t) const

{

  typedef typename std::tuple_element<

    index, std::tuple<types…>>::type nexttype;

  hash_impl<index-1, types…> next;

  size_t b = std::hash<nexttype>()(std::get<index>(t));

  return next(hash_combiner(a, b), t);

}

};


template<class…types>

struct hash_impl<0, types…>
```

```
{
size_t operator()(size_t a, const std::tuple<types…>& t) const
{
  typedef typename std::tuple_element<0,
    std::tuple<types…>>::type nexttype;
  size_t b = std::hash<nexttype>()(std::get<0>(t));
  return hash_combiner(a, b);
}
};

namespace std {
  template<class…types>
  struct hash<std::tuple<types…>> {
    size_t operator()(const std::tuple<types…>& t) const {
      const size_t begin =
    std::tuple_size<std::tuple<types…>>::value-1;
    return hash_impl<begin, types…>()(1, t);
    }
  };
}
```

Bibliography

1. "Side effect (computer science)", Wikipedia, http://en.wikipedia.org/w/index.php?title=Side_effect_(computer_science)&oldid=650829249
2. Boost MPL, http://www.google.com/search?btnI=I'm+Feeling+Lucky&q=boost%20mpl
3. Boost Fusion 2, http://www.google.com/search?btnI=I'm+Feeling+Lucky&q=boost%20fusion
4. Abrahams, D., Gurtovoy, A., C++ Template Metaprogramming, http://amzn.com/0321227255
5. Milewski, Bartosz, "Functional Data Structures in C++: Lists", http://bartoszmilewski.com/2013/11/13/functional-data-structures-in-c-lists/

Books from the Author