Lab: Half-Open Ranges with Function Templates

Learning Outcomes

- Gain experience with function templates
- Gain experience with C++ standard library's half-open ranges
- Gain practice in debugging memory leaks and errors using Valgrind
- Reading, understanding, and interpreting C++ unit tests written by others

Overview

The purpose of this laboratory is gain experience with function templates, half-open ranges, pointers. The goal is to implement several templated functions that perform operations on half-open ranges such as removing elements, replacing elements, searching for elements, copying one range to another range, etc. Many of these functions mimic how generic algorithms in the C++ standard library work.

Task

Your task can be summed up as follows:

1. Each function template to be defined is tested by a unit test function in driver `ft-driver.cpp`. This document doesn't explain the specs of each function template. Instead, you should understand the spec of a function template from the corresponding unit test and the correct output generated by the unit test.
2. Declare, define, and test each function template individually by commenting out everything except the unit test function exercising the function template.
    - Because of the two-phase compilation policy used by C++ compilers to instance and compile function templates, the necessary function templates must be declared and defined in header file `ft.hpp`. There is one caveat:

      > `ft.hpp` ***must only include*** `<iostream>`***. Don't include any other header file!!!***

    - Each function template must be defined using a single function - that is, you should not resort to overloading function templates to successfully complete the test. Instead, think of expanding the number of template type parameters from one to two. This is an important point because the final test will require you to implement similar use cases with a single function template.
    - This is an example of a function template with one template type parameter:

      ```
      template <typename T>
      T const& compare(T const& lhs, T const& rhs) {
        if (lhs < rhs) return -1;
        if (rhs < lhs) return 1;
        return 0;
      }
      ```

    - This is an example of a function template with two template type parameters:

```
template <typename Base, typename Exponent>
Base power(Base b, Exponent e) {
  return std::pow(b, e); // requires inclusion of <cmath>
}
```

- If you're not sure of half-open ranges, review the topic here before proceeding with defining them. This section describes the use of half-open ranges in relation to C++ standard library container `std::vector`. Compact pointer expressions are commonly employed when iterating through half-open ranges. You can review this topic here.
- The C++-way of implementing a function template that iterates over a half-open range is illustrated here:

```
template <typename Iter>
void foo(Iter first, Iter last) {
  while (first != last) {
    // 1) access the value using dereference operator: *first
    // 2) do something with the value
    // 3) iterate to next element:
    ++first;
  }
}
```

  It is crucial that you do not think of using the subscript operator `[]` to access the value pointed to by iterator `first`. This is because many C++ standard library iterators do not overload the subscript operator; they *only* overload dereference operator `*`. This is the C++-way of iterating over half-open ranges and you don't have other alternatives.

  > ***Since many standard library iterators don't overload the subscript operator, don't use the subscript operator to access the value pointed to by an iterator. Instead, use the dereference operator which is overloaded by all C++ standard library iterators. It is possible to have questions in the final test that prevent the use of subscript operator. Therefore, obtain the necessary practice and knowledge today by implementing function templates that iterate over half-ranges using only the dereference operator.***

3. After implementing the function template, run the test and make sure it generates the correct output. Compare your function's output with the correct program output for that specific test in `ft-output.txt`.
4. Continue this approach until all the tests succeed. Your program's output should be exactly equivalent to the entire contents of `ft-output.txt`.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header and source files

Submit only file `ft.hpp`. There is no other file to submit.

Compiling, executing, and testing

Download `sllist-driver.cpp`, `makefile`, and correct output file `output.txt` generated by the unit tests. Run `make` with the default rule [`$ make`] to bring program executable `sllist.out` up to date. Or, directly test your implementation by running `make` with target `test`: `$ make test`. If the `diff` command in the `test` rule is not silent, then one or more of your function definitions is incorrect and will require further work.

Submission Details

Please read the following details carefully and adhere to all requirements to avoid unnecessary deductions.

Header file and documentation details

You will be submitting file `ft.hpp`. This file will contain the declarations and definitions of the function templates you must implement in the following format:

```cpp
//--------------------------------------------------------------------------
#ifndef FT_HPP
#define FT_HPP
//--------------------------------------------------------------------------
#include <iostream>
// don't include any other header files!!!
// otherwise, your code won't compile.

namespace hlp2 {
// DECLARE (not define!!!) and DOCUMENT in ALPHABETIC ORDER the function
// templates you'll be defining. There are *13* function templates to be
// declared and defined.

// I'm providing the declaration and documentation for swap here:
/***************************************************************************
/
/*!
\brief
 Swaps two objects. There is no return value but the values in
 the two objects are swapped in place.

\param left
  Reference to the first object to swap.

\param right
 Reference to the second object to swap.
*/
/**************************************************************************/
template <typename T> void swap(T &left, T &right);

// Provide DEFINITIONS for each function template declared above ...
}

#endif
//--------------------------------------------------------------------------
```

Compiling, executing, and testing

Download `ft-driver.cpp`, `student.hpp` [containing definition of type `Student`], and output file `ft-output.txt` containing the correct output for the unit tests in the driver. Follow the steps from the previous lab to refactor a `makefile` that can compile, link, and test your program.

Submission and automatic evaluation

1. In the course web page, click on the appropriate submission page to submit the necessary file.
2. Please read the following rubrics to maximize your grade. Your submission will receive:
   - F grade if your submission doesn't compile with the full suite of `g++` options.
   - F grade if your submission doesn't link to create an executable.
   - Your implementation's output doesn't match correct output of the grader (you can see the inputs and outputs of the auto grader's tests). The auto grader will assign 50% of the grade based on the input and output files given to you. The remaining 50% of the grade will be awarded based on the additional tests implemented by the auto grader.
   - The auto grade will provide a proportional grade based on how many incorrect results were generated by your submission. $A+$ grade if your output matches correct output of auto grader.
   - A deduction of one letter grade for each missing documentation block in your submissions. Each source file must have **one** file-level documentation block and a function-level documentation block for each defined function. A teaching assistant will physically read submitted source files to ensure that these documentation blocks are authored correctly. Each missing or incomplete or copy-pasted (with irrelevant information from some previous assessment) block will result in a deduction of a letter grade. For example, if the automatic grader gave your submission an $A+$ grade and one documentation block is missing, your grade will be later reduced from $A+$ to $B+$. Another example: if the automatic grade gave your submission a C grade and the two documentation blocks are missing, your grade will be later reduced from C to F.

Half-open ranges

When an array is passed to a function, the array's base address is passed by value. This makes the transfer of arrays between functions extremely efficient since copies of individual array elements don't have to be passed to the function. The function can use the array's base address in conjunction with integer offsets to iterate through every array element. To prevent too few elements from being accessed or accessing out-of-bounds elements, the array's size must also be passed. Since strings are terminated by the null character `'\0'`, functions that process strings don't require the array's size to be passed.

The function parameter initialized with the base address can be declared using either array or pointer notation. Consider the definition of function `accumulate` with array notation:

```cpp
int accumulate(int const arr[], int size) { // array notation
  int sum {0};
  for (int i {0}; i < size; ++i) {
    sum += arr[i]; // using subscript operator
  }
  return sum;
}
```

The same function can be defined with pointer notation:

```cpp
int accumulate(int const *arr, int size) { // pointer notation
  int sum {0};
  for (int i {0}; i < size; ++i) {
    sum += *(arr+i); // using pointer offset
  }
  return sum;
}
```

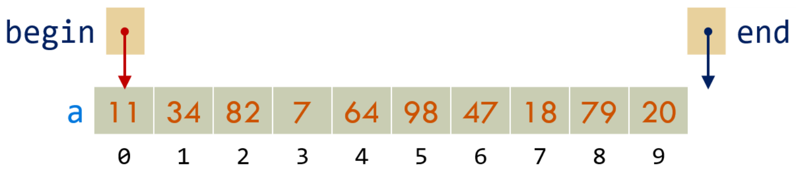
Function `accumulate` is typically used to compute the sum of an entire array:

```cpp
int const SIZE {10};
int a[SIZE] = {11, 34, 82, 7, 64, 98, 47, 18, 79, 20};
std::cout << "sum: " << accumulate(a, SIZE) << '\n';
```

The function can also be used to compute the sum of a range of elements that comprise a slice of the array. For example, the array can be implicitly sliced into a front half slice, a back half slice, and a third slice consisting of the middle six elements:

```cpp
// sum of first half
std::cout << "sum: " << accumulate(a, SIZE/2) << '\n';
// sum of second half
std::cout << "sum: " << accumulate(a+SIZE/2, SIZE/2) << '\n';
// sum of middle six elements
std::cout << "sum: " << accumulate(a+2, 6) << '\n';
```

Ranges of elements that might, but are not required to, specify all elements of an array can be more flexibly specified using *half-open ranges*. A half-open range is defined so that it includes the element used as the beginning of the range but excludes the element used as the end. This concept is described by the mathematical notation for half-open ranges as $[begin, end)$ where pointers `begin` and `end` point to the elements at the beginning and end of the range, respectively. The range consisting of all 10 elements of array `a` is defined with pointer `begin` pointing to the first element of `a` while pointer `end` is a *past-the-end* pointer pointing to the element after the last array element.



Function `accumulate` is redefined to incorporate the concept of half-open ranges:

```cpp
int accumulate(int const *begin, int const *end) {
  int sum {0};
  while (begin < end) {
    sum += *begin++;
  }
  return sum;
}
```

Parameters `begin` and `end` in the above functions represent a range of array elements. Parameter `begin` points to the first element in the range while parameter `end` is a *past-the-end* pointer pointing to the element after the last element in the range. Thus, `begin` and `end` define a *half-open range* `[begin, end)` that includes the first element (pointed to by `begin`) but excludes the last element (pointed to by `end`).

The range-based version of `accumulate` can be used to compute different ranges of elements of array `a`:

```
// sum of all array elements
std::cout << "sum: " << accumulate(a, a+SIZE) << '\n';
// sum of first half
std::cout << "sum: " << accumulate(a, a+SIZE/2) << '\n';
// sum of second half
std::cout << "sum: " << accumulate(a+SIZE/2, a+SIZE) << '\n';
// sum of middle six elements
std::cout << "sum: " << accumulate(a+2, a+8) << '\n';
```

A half-open range has two advantages. First, you can define a simple end criterion for loops that iterate over the elements: they start at `begin` and simply continue as long as `end` is not reached. Second, special handling for empty ranges is avoided. For empty ranges, `begin` is equal to `end`. The following code fragment illustrates the function for an empty range:

```
std::cout << "sum: " << accumulate(a, a) << '\n'; // sum should be zero
```

Although half-open ranges provide a flexible interface, they're also dangerous. The caller must ensure that the first two arguments define a valid range. A range is valid if the end of the range is reachable from the beginning by incrementing the pointer to successively point to each array element. This means that it is up to the caller to ensure that both pointers point to elements in the same array and that the beginning is at a lower memory address than the end. Otherwise, the behavior is undefined, and endless loops or out-of-bounds memory accesses may result.

Using `begin()` and `end()`

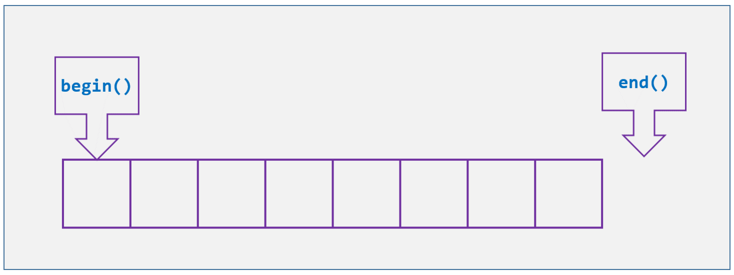
We've used subscripts to access elements of a `vector` and individual `char` elements of a `string`. We can get pointers to first and last members of `vector`s and `string`s using member functions `front` and `back`, respectively. These pointers can be used to access individual elements of `vector`s and `string`s:

```
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};

// access char elements of a string using pointers
char *ptr_char = &cities[0].front(), *ptr_last_char = &cities[0].back();
while (ptr_char <= ptr_last_char) {
  std::cout << *ptr_char++;
}
std::cout << "\n";

// access string elements of a vector using pointers:
for (std::string *p = &cities.front(); p <= &cities.back(); ++p) {
  std::cout << *p << "\n";
}
```

The C++ standard library provides the concept of iterators that abstract the interface of ordinary pointers. Just like pointers, iterators give us indirect access to an object. As with pointers, we can use operators `++` and `--` to step forward and backward, respectively. We can use operators `==` and `!=` to determine whether two iterators represent the same position. We can assign iterators using operator `=`. We'll study iterators in more detail at a later date in the semester. For now, we'll use iterators to traverse `string`s since they simply encapsulate a `char *`. Likewise, we use iterators to traverse `vector`s since they encapsulate an ordinary pointer.

All container classes provide the same member functions that enable them to use iterators to navigate over their elements. The most important of these functions are as follows:

- `begin` returns an iterator that points to the first element in the container.
- `end` returns an iterator that points to the position after the last element of the container.



Thus, as shown in the picture, member functions `begin` and `end` define a *half-open range* that *includes the first element* but *excludes the last*. A half-open range has two advantages:

- There is a simple criterion to terminate loops that iterate over elements of a container. Loops simply continue as long as function `end` is not reached.
- Empty containers will not require special handling since function `begin` is equal to function `end`.

Rather than using subscripts, the following example demonstrates the use of iterators to traverse the elements of a `string` and convert every alphabetic character to uppercase:

```
#include <cctype> // for std::isalpha, std::islower, std::toupper

std::string s {"Runtime error!!!"};
for (std::string::iterator it = s.begin(); it != s.end(); ++it) {
  if (std::isalpha(*it) && std::islower(*it)) {
    *it = std::toupper(*it);
  }
}
```

Every container defines two iterator types:

- `container::iterator` is provided to iterate over elements in read/write mode.
- `container::const_iterator` is provided to iterator over elements in read-only mode.

The following example demonstrates the use of read-only iterators to traverse the elements of a `vector`:

```
std::vector<std::string> cities {"Seattle", "San Jose",
                                 "Singapore", "Shanghai"};

// iterate over the elements of container cities: vector::begin() is
// overloaded to return either read/write or read-only iterator. Because
// cit has type read-only iterator, cities.begin() will return read-only
// iterator
for (std::vector<std::string>::const_iterator cit = cities.begin();
     cit != cities.end(); ++cit) {
  std::cout << *cit << "\n";
}

// here, we explicitly call cbegin() and cend()
for (std::vector<std::string>::const_iterator cit = cities.cbegin();
     cit != cities.cend(); ++cit) {
  std::cout << *cit << "\n";
}
```

Compact pointer expressions

Compact pointer expressions involve the use of indirection operator `*`, unary increment `++` and decrement `--` operators with pointer operands. When compact pointer expressions are used, the compiler may potentially produce assembly code that is smaller in size and runs faster because most modern microprocessors have addressing modes or instructions that combine indirection with increments and decrements. The precedence and associativity of `++`, `--`, `*`, and `&` operators are:

Precedence	Operator	Description	Associativity
1	`++` `--`	postfix increment and decrement	left-to-right

2	`++` `--`
	`*`
	`&`
		prefix increment and decrement
		dereference
		address-of
			right-to-left

Suppose a pointer variable `p` points to an array element. The following table lists four forms of compact pointer expressions:

Expression	Operation	Affects	Reads as
`*p++`	post increment	pointer	`*(p++)`
`*p--`	post decrement	pointer	`*(p--)`
`*++p`	pre increment	pointer	`*(++p)`
`*--p`	pre decrement	pointer	`*(--p)`

Expressions from the table above modify the pointer and not the pointer's object. Pointers may point to any C/C++ data type and typically access elements of an array. These expressions improve a program's execution speed if the machine implements fetch-and-increment instructions.

Again, suppose a pointer variable `p` points to an array element. The following table lists a second set of four compact pointer expressions:

Expression	Operation	Affects	Reads as
`++*p`	pre increment	object	`++(*p)`
`--*p`	pre decrement	object	`--(*p)`
`(*p)++`	post increment	object	`(*p)++`
`(*p)--`	post decrement	object	`(*p)--`

Expressions from the second table affect the pointer's object rather than the pointer. These objects may be simple data types like integers, characters, or array elements of simple types. Note that a pointer to a structure, union, or function *cannot* be used with these expressions.

Consider the following code fragment:

```
char str[] = "SeaToShiningC";
char *p     = str+5;
```

The following table illustrates the effects of compact pointer expressions. The rows are not cumulative - for each expression, pointer `p` is initialized as: `char *p = str+5;`.

Expression	Result	Pointer pointing to	Resultant string
`*p++`	`'S'`	`'h'`	`"SeaToShiningC"`
`*p--`	`'S'`	`'o'`	`"SeaToShiningC"`
`*++p`	``` `'h' ```	`'h'`	`"SeaToShiningC"`
`*--p`	`'o'`	`'o'`	`"SeaToShiningC"`
`++*p`	`'T'`	`'T'`	`"SeaToThiningC"`
`--*p`	`'R'`	`'R'`	`"SeaToRhiningC"`
`(*p)++`	`'S'`	`'T'`	`"SeaToThiningC"`
`(*p)--`	`'S'`	`'R'`	`"SeaToRhiningC"`

Applications of compact pointer expressions

Standard C library function `strcpy` is declared in `<cstring>` as:

```
char *strcpy(char *dst, char const *src);
```

Function `strcpy` copies the string whose first element is pointed to by `src` to the destination character array whose first element is pointed to by `dst`, assuming the destination array is large enough to hold the characters in string `src`. Further, the function returns a pointer to the first element of the destination array. The following code fragment illustrates sample use cases of `strcpy`:

```
#include <cstring>  // for strcpy
#include <iostream> // for printf

char str1[81], str2[] = "today is a good day";
std::strcpy(str1, str2);
std::cout << "str1: " << str1 << '\n'; // prints: today is a good day
std::strcpy(str1, "tomorrow will be a better day");
std::cout << "str1: " << str1 << '\n'; // prints: tomorrow is a better day
```

Here, the use of compact pointer expressions is examined by writing different versions of function `strcpy` called `my_strcpy`. The first version uses array subscripting:

```
// version 1: uses array subscripting
char *my_strcpy(char *dst, char const *src) {
  int i {0};
  // iterate through each element of array whose first element is pointed
  // to by pointer src until the null character is encountered.
  // copy each encountered element from array whose first element is pointed
  // to by src to the array whose first element is pointed to by dst
  while (src[i] != '\0') {
    dst[i] = src[i];
    ++i;
  }
  dst[i] = '\0';
  return dst;
}
```

The second version continues to use array subscripting but writes a simpler `while` condition using the knowledge that null character `'\0'` has decimal value 0:

```
// version 2: uses array subscripting
char *my_strcpy(char *dst, char const *src) {
  int i {0};
  while (src[i]) {
    dst[i] = src[i];
    ++i;
  }
  dst[i] = '\0';
  return dst;
}
```

The third version again uses array subscripting but is simpler than the previous version. This version uses the insight that an assignment expression evaluates to the value written to the lvalue operand to left of the assignment operator:

```
// version 3: uses array subscripting
char *my_strcpy(char *dst, const char *src) {
  int i = 0;
  // the while expression will copy characters from source string to
  // destination string including the null character.
  // when the null character is copied, the while condition evaluates to
  // zero (or false), and the loop terminates.
  while ((dst[i] = src[i])) { // extra parantheses to avoid naggy compiler
    ++i;
  }
  return dst;
}
```

For the fourth version of function `strcpy`, we replace the subscript operator `[]` with pointer offsets and dereference operator `*`:

```
// version 4: uses pointer offsets and dereference operator
char *my_strcpy(char *dst, const char *src) {
  int i {0};
  // extra parantheses to avoid naggy compiler
  while ((*(dst+i) = *(src+i))) { ++i; }
  return dst;
}
```

The previous version provides a path to replacing pointer offsets with pointers that point to appropriate locations in the source and destination strings.

```
// version 5: uses deference and increment operators
char *my_strcpy(char *dst, const char *src) {
  // since the function must return a pointer to the first element in the
  // destination string, a copy of the address is required
  char *tmp {dst};
  while ((*dst = *src)) { // extra parantheses to avoid naggy compiler
    ++src; // pointer to next character in source string
    ++dst; // pointer to next element in destination string
  }
  return tmp;
}
```

The sixth and final version of function `my_strcpy` uses compact pointer expressions:

```
// version 6: uses compact pointer expressions
char *my_strcpy(char *dst, const char *src) {
  char *tmp {dst};
  while ((*dst++ = *src++)) { // extra parantheses to avoid naggy compiler
  }
  return tmp;
}
```