Swapping in C++

```
#include <utility>
#include <iostream>
#include <string>
using namespace std;


int main() {
    int a = 17, b = 42;
    cout << a << ' ' << b << endl;
    a, b = b, a;
    cout << a << ' ' << b << endl;

}
```

```
??
```

Swapping in C++ - cont.

```
#include <utility>
#include <iostream>
#include <string>
using namespace std;


int main() {
    int a = 17, b = 42;
    cout << a << ' ' << b << endl;
    a, b = b, a;
    cout << a << ' ' << b << endl;

}
```

```
17 42
17 42
```

- Sadly compiles in C++, but <u>doesn’t mean what it does in Python</u>.

- C++ converts it into: a, (b = b), a;
  - The commas are just separators allowing multiple “sub” expressions in a larger expression, with the last one being the “value”.

- So the three expressions are evaluated in turn. Compiler should warn of unused a, twice.

Swapping in C++ - cont.

```cpp
#include <utility>
#include <iostream>
#include <string>
using namespace std;

template <typename T>
void mySwap(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}

int main() {
    int a = 17, b = 42;
    cout << a << ' ' << b << endl;
    a, b = b, a;
    cout << a << ' ' << b << endl;


    swap(a, b);
    cout << a << ' ' << b << endl;
    mySwap(a, b);
    cout << a << ' ' << b << endl;


}
```

```
17 42
17 42
42 17
17 42
```

The C++ Standard Template Library (STL)

- Utility library
  - Types (e.g. pair)
  - Functions (e.g. make_pair(), swap(), etc.)
- **Container classes (e.g. vector, list, queue, map, set, etc.)**
  - **All support the begin() and end() methods for a half-open range**
- Functional library (e.g. functor, aka function object)
- Algorithms (e.g. find, sort, etc.)

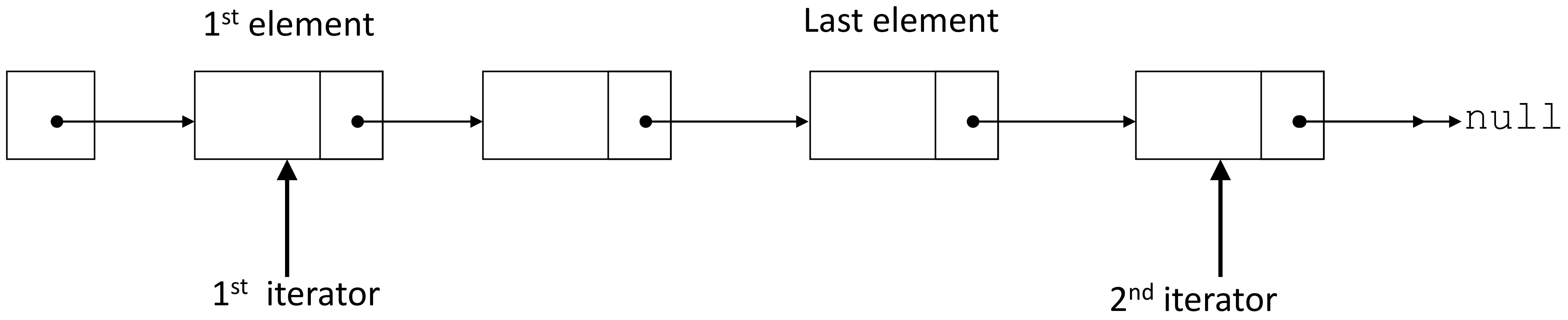
STL containers – half open range

- **A range of elements is a sequence of elements denoted by two iterators:**
  - The **first iterator points to the first element** in the range
  - The **second iterator** points to one-after the end of the range (the **element to which the second iterator points is not included in the range**).

  - This is sometimes referred to as **half open range**.

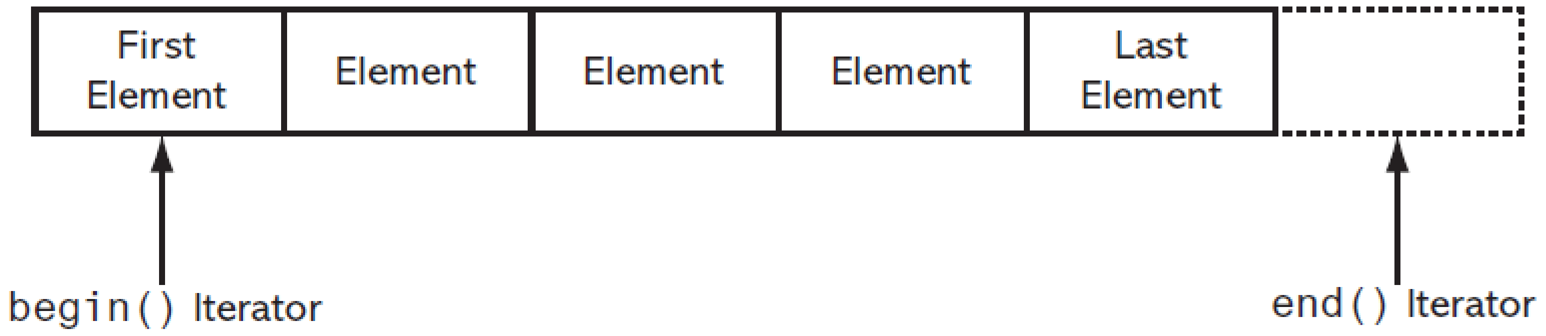
  - May be used with any STL container class (e.g. vector, list, map, etc.)

Half open range – cont.

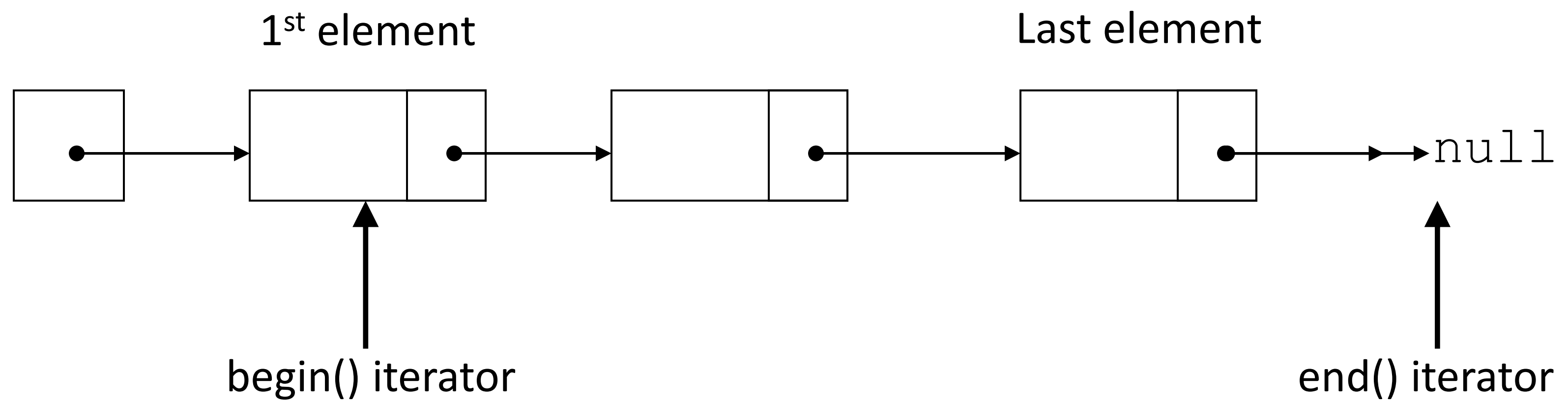


The begin() and end() methods with std::vector

- All of the STL containers have a **`begin()`** member function that returns an iterator pointing to the container's first element.
- All of the STL containers have a **`end()`** member function that returns an iterator pointing to the position *after* the container's last element.



The begin() and end() methods with std::list



The begin() and end() methods – cont.

- You can use the **auto** keyword to simplify the return type from begin() and end() (return is usually of type class_name::iterator)
- Example:

```
vector<string> names = {"Sarah", "William", "Alfredo"};
for( auto it = names.begin(); it!=names.end(); ++it) cout << *it;
```

- <u>Not recommended when</u> `the loop variable is an int or size_t, i.e not recommended in:`

```
for( size_t i=0; i<names.size(); ++i) cout < names[i];
```

Character arrays (c-strings)

- Stored as an array of type `char`

  e.g. `char mychararr[] = “Hello”;`

  is actually stored with the null terminator, `\0`, at the end:

H	e	l	l	o	\0

- In the “Hello” string example, the array is of size 6 bytes, and is sometimes referred to as a “null terminated character string”.
- The null character is very important for many of the string manipulation functions.

Example

```
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {

    char array[] = "Hello world, this is CS2124 !!";


}
```

How do we create a vector and initialize it with content of “array”

Example – cont.

```
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {

    char array[] = "Hello world, this is CS2124 !!";

    // Initialize a vector with content of this char array.
    int len = sizeof(array);
    vector<char> vc(array, array + len);
    for (char c : vc) cout << c;
    cout << endl;




}
```

How do we create a list and initialize it with content of the vector?

Example – cont.

```
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {

    char array[] = "Hello world, this is CS2124 !!";

    // Initialize a vector with content of this char array.
    int len = sizeof(array);
    vector<char> vc(array, array + len);
    for (char c : vc) cout << c;
    cout << endl;

    // Initialize a list with content of the vector
    list<char> lc(vc.begin(), vc.end());
    for (char c : lc) cout << c;
    cout << endl;



}
```

```
Hello world, this is CS2124 !!
Hello world, this is CS2124 !!
```

The C++ Standard Template Library (STL)

- Utility library
  - Types (e.g. pair)
  - Functions (e.g. make_pair(), swap(), etc.)
- Container classes (e.g. vector, lists, queue, map, set, etc.)
  - All support the begin() and end() methods for a half-open range
- **Functional library (e.g. functor, aka function object)**
- **Algorithms (e.g. find, sort, etc.)**
  - The STL provides a number of algorithms, implemented as function templates, in the `<algorithm>` header file.
  - These functions perform various operations on ranges of elements.

Categories of Algorithms in the STL

- Min/max algorithms
- Sorting algorithms
- **Search algorithms**
- Read-only sequence algorithms
- Copying and moving algorithms
- Swapping algorithms
- Replacement algorithms
- Removal algorithms
- Reversal algorithms
- Fill algorithms
- Rotation algorithms
- Shuffling algorithms
- Set algorithms
- Transformation algorithm
- Partition algorithms
- Merge algorithms
- Permutation algorithms
- Heap algorithms
- Lexicographical comparison algorithm

STL Algorithms – linear search

```
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

char* myFind(char* start, char* stop, char target) {
    for (char* p = start; p < stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}

int main() {

    char array[] = "Bjarne Stroustrup";
    int len = 17;

    vector<char> vc(array, array + len);
    list<char> lc(vc.begin(), vc.end());

    cout << *find(array, array+len, 'S') << endl;

    cout << *find(lc.begin(), lc.end(), ‘r’) << endl;

    cout << *myFind(array, array+len, ‘j’) << endl;
}
```

```
S
r
j
```

“find()” is part of the STL algorithms lib.

STL Algorithms – generic linear search

```
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

template <typename T, typename U>
T myFind(T start, T stop, U target) {
    for (T p = start; p != stop; ++p) {
        if (*p == target) {
            return p;
        }
    }
    return stop;
}

int main() {

    char array[] = "Bjarne Stroustrup";
    int len = 17;

    vector<char> vc(array, array + len);
    list<char> lc(vc.begin(), vc.end());

    cout << *find(array, array+len, 'S') << endl;

    cout << *find(lc.begin(), lc.end(), 'r') << endl;

    cout << *myFind(array, array + len, 'j') << endl;

}
```

```
S
r
j
```

Plugging Your Own Functions into an Algorithm – predicates

- Many of the function templates in the STL are designed to accept a "**predicate**"
- This allows you to "plug" one of your own functions into the algorithm. Example:

```
template <typename T, typename U>
T myFind_if(T start, T stop, U predicate) {
    for (T p = start; p != stop; ++p) {
        if (predicate(*p)) {
            return p;
        }
    }
    return stop;
}
```

This is a unary predicate

- A predicate may be
  - Function pointer
  - Function object
  - Lambda expression

STL Algorithms - search with fun. Ptr. predicates

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <typename T, typename U>
T myFind_if(T start, T stop, U predicate) {
    for (T p = start; p != stop; ++p) {
        if (predicate(*p)) {
            return p;
        }
    }
    return stop;
}

bool isOdd(int n) { return n % 2 != 0; }

int main() {

    int a[]{ 90, 30, 23, 68, 4 };

    int temp = *find_if(a, a + 5, isOdd);
    cout << “Element is: " << temp << endl;

    temp = *myFind_if(a, a + 5, isOdd);
    cout << “Element is: " << temp << endl;


}
```

```
Element is: 23
Element is: 23
```

Function Objects

- A **function object** is an object that acts like a function.
  - It can be called
  - It can accept arguments
  - It can return a value

- Function objects are also known as ***functors***

Function Objects as predicates

- To create a function object, you write a class that overloads the ( ) operator.

```
#include <iostream>
using namespace std;


class BiggerThan{
public:
    bool operator() (int a, int b) {
        return a > b;
    }
};

int main() {

    int x = 15, y = 22;
    BiggerThan bt;
    cout << "Invoking bt(x,y):  " << (bt(x, y)? "true": "false") <<
        endl;
}
```

```
Invoking bt(x,y):  false
```

STL Algorithms – search with functor predicates

```cpp
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

template <typename T, typename U>
T myFind_if(T start, T stop, U predicate) {
    for (T p = start; p != stop; ++p) {
        if (predicate(*p)) {
            return p;
        }
    }
    return stop;
}

bool isOdd(int n) { return n % 2 != 0; }
struct IsEven {
    bool operator() (int n) const { return n % 2 == 0; }
};
struct IsMultiple {
    IsMultiple(int n) : divisor(n) {}
    bool operator() (int n) { return n % divisor == 0; }
    int divisor;
};

int main() {

    int a[]{ 90, 30, 23, 68, 4 };

    int temp = *find_if(a, a + 5, isOdd);
    cout << "Element is: " << temp << endl;

    temp = *myFind_if(a, a + 5, isOdd);
    cout << "Element is: " << temp << endl;

    IsEven isEven; // functor
    cout << isEven(17) << endl;
    cout << *find_if(a, a + 5, isEven) << endl;

    IsMultiple multOf7(7);
    find_if(a, a + 5, multOf7);
    cout << *find_if(a, a + 5, IsMultiple(17)) << endl;
}
```

```
Element is: 23
Element is: 23
0
90
68
```

- **Functor**
  - Also known as "function object".
  - An object whose class implements the function call operator.
- Anonymous Function Objects as predicates

Lambda Expressions

- A lambda expression is a compact way of creating a function object without having to write a class declaration.

- It is an expression that contains only the logic of the object's `operator()` member function.

- When the compiler encounters a lambda expression, it automatically generates a function object in memory, using the code that you provide in the lambda expression for the `operator()` member function.

Lambda Expressions

- General format:

  `[](`*`parameter list`*`) { `*`function body`*` }`

- The `[]` is known as the lambda introducer. It marks the beginning of a lambda expression.
- *`parameter list`* is a list of parameter declarations for the function object's `operator()` member function.
- *`function body`* is the code that should be the body of the object's `operator()` member function.

Lambda Expressions

- Example: a lambda expression for a function object that computes the sum of two integers:

  ```
  [](int a, int b) { return x + y; }
  ```

Lambda Expressions

- Example: a lambda expression for a function object that determines whether an integer is even is:

```
[](int x) { return x % 2 == 0; }
```

Lambda Expressions

- Example: a lambda expression for a function object that takes an integer as input and prints the square of that integer:

```
[](int a) { cout << a * a << " "; }
```

Lambda Expressions

- When you call a lambda expression, you write a list of arguments, enclosed in parentheses, right after the expression.
- For example, the following code snippet displays 7, which is the sum of the variables x and y:

```
int x = 2;
int y = 5;
cout << [](int a, int b) {return a + b;}(x, y) << endl;
```

Lambda Expressions

- The following code segment counts the even numbers in a `vector`:

```
// Create a vector of ints.
vector<int> v = { 1, 2, 3, 4, 5, 6, 7, 8 };

// Get the number of elements that are even.
int evenNums = count_if(v.begin(), v.end(), [](int x) {return x % 2 == 0;});

// Display the results.
cout << "The vector contains " << evenNums << " even numbers.\n";
```

Lambda Expressions

- Because lambda expressions generate function objects, you can assign a lambda expression to a variable and then call it through the variable's name:

```
auto sum = [](int a, int b) {return a + b;};
int x = 2;
int y = 5;
int z = sum(x, y);
```

STL Algorithms – search with lambda expressions

```cpp
#include <vector>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {

    int a[]{ 90, 30, 23, 68, 4 };

    // using lambda expressions
    find_if(a, a + 5, [](int n) { return n % 2 == 0; });
    find_if(a, a + 5, [](int n) -> bool { return n % 2 == 0; });

    [] { cout << "lambda\n"; }();

    auto func = [] { cout << "lambda\n"; };
    func();

}
```

```
lambda
lambda
```