The C++ Standard Template Library (STL)

A C++ standard library that **uses generic types** (i.e. templated). It provides:

- **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 pair type

Natural to pair data

- (x,y) coordinates
- product description/price
- student/grade
- name/ID

```
std::pair<string, double> product;
std::pair<int, int> coord;
std::pair<string, char> mark;
std::pair<string, string> employee;
```

```
std::pair<type1, type2>  my_pair;
```

The pair type

```cpp
#include <utility>  // pair, make_pair
#include <iostream>
#include <string>
using namespace std;

pair<int, string> foo() {
    pair<int, string> result(42, "the answer");
    return result;
}

// c++14
auto bar() {
    return make_pair(42, "the answer");
}

int main() {
    pair<int, string> result = foo();
    cout << result.first << ": " << result.second << endl;

    // c++11
    auto result2 = foo();
    cout << result2.first << ": " << result2.second << endl;



}
```

```
42: the answer
42: the answer
```

The use of auto for function return type is a **c++14** feature.

C++ is a statically typed language:

- **auto** May be used to declare an initialized variable, OR
- as a return type

The pair type – structured bindings

```cpp
#include <utility>  // pair, make_pair
#include <iostream>
#include <string>
using namespace std;

pair<int, string> foo() {
    pair<int, string> result(42, "the answer");
    return result;
}

// c++14
auto bar() {
    return make_pair(42, "the answer");
}

int main() {
    pair<int, string> result = foo();
    cout << result.first << ": " << result.second << endl;

    // c++11
    auto result2 = foo();
    cout << result2.first << ": " << result2.second << endl;


    // c++17: structured binding
    auto [a, b] = foo();
    cout << a << ": " << b << endl;

    auto [x, y] = bar();
    cout << x << ": " << y << endl;

}
```

```
42: the answer
42: the answer
42: the answer
42: the answer
```

The use of auto for function return type is a **c++14** feature

`Requires C++17`

The pair type – cont.

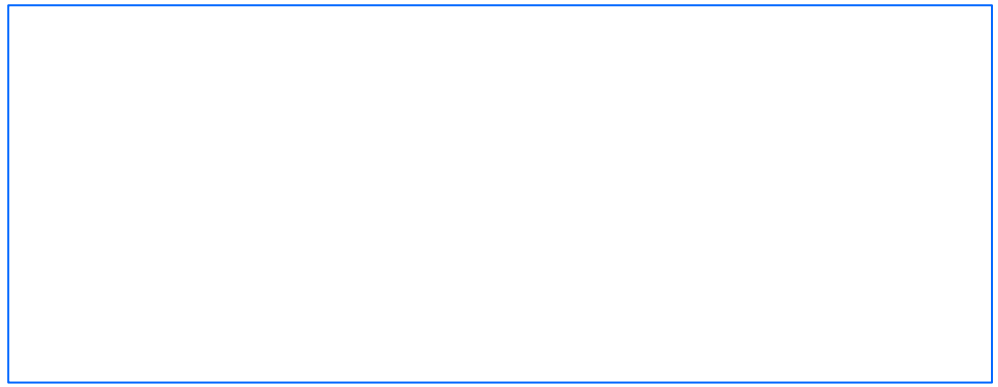
```cpp
#include <utility>
#include <iostream>
#include <string>
using namespace std;

//pair<int, int> returnsTwo() {
auto returnsTwo() {
    //     return 6, 28;
    return make_pair(6, 28);
}


int main() {

    pair<int, int> result = returnsTwo();
    cout << result.first << ' ' << result.second << endl;

    auto result2 = returnsTwo();
    cout << result2.first << ' ' << result2.second << endl;

    // structured [un]binding
    auto [x, y] = returnsTwo();
    cout << x << ' ' << y << endl;
}
```

```
6 28
6 28
6 28
```

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.

```cpp
#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
```

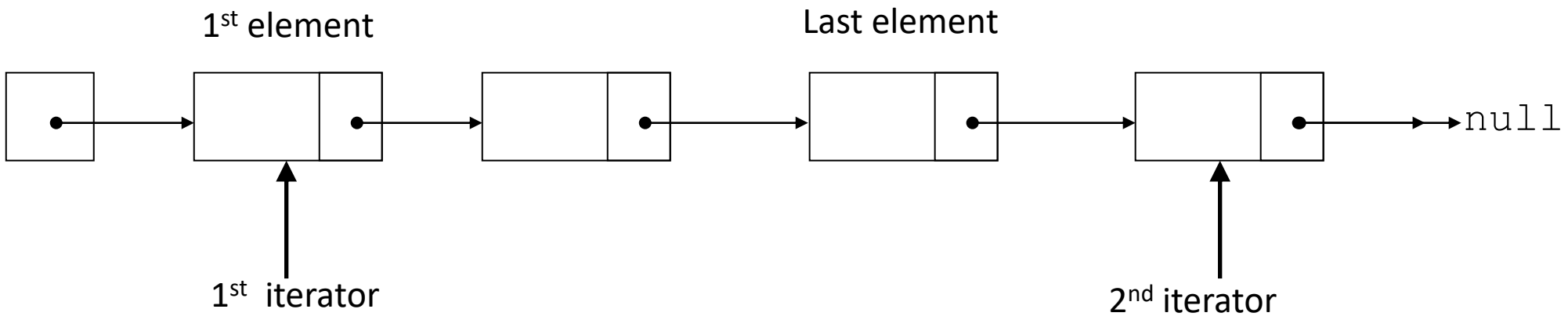
The C++ Standard Template Library (STL)

- Utility library
  - Types (e.g. pair)
  - Functions (e.g. make_pair(), swap(), etc.)
- **Container classes (e.g. vector, 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.)

STL containers – half open range

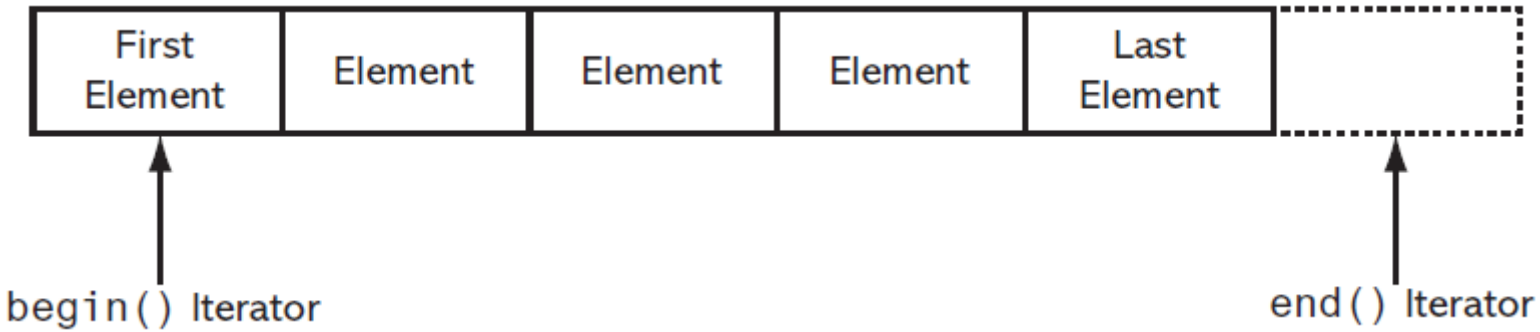
- **A range of elements is a sequence of elements denoted by two iterators:**
  - The **first iterator points to the first element** in the range
  - The **second iterator** points to the end of the range (the **element to which the second iterator points is not included in the range**).
  - This is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

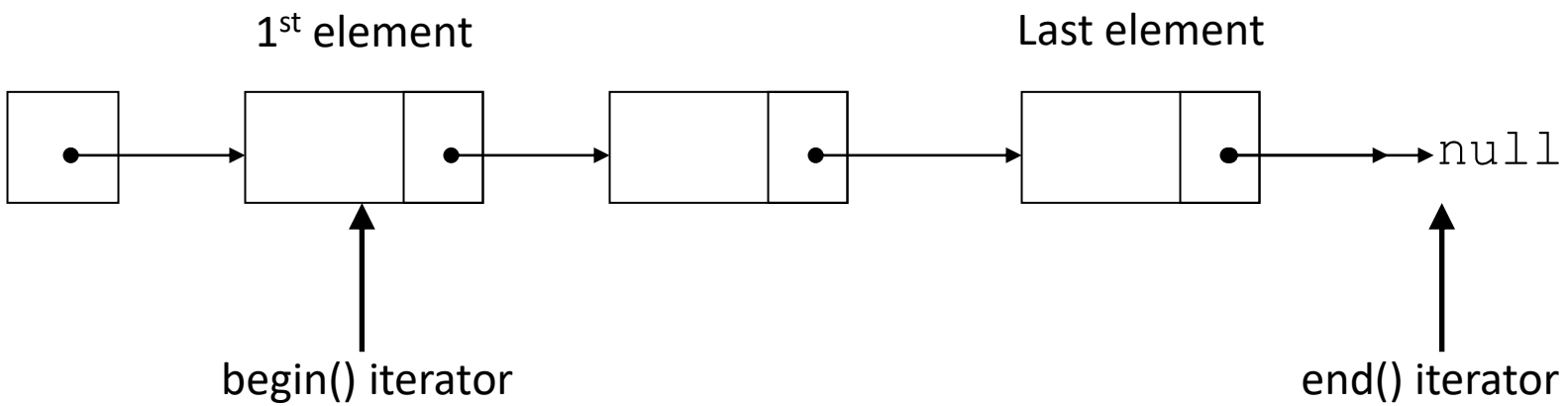


The begin() and end() methods with std::vector

- 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:

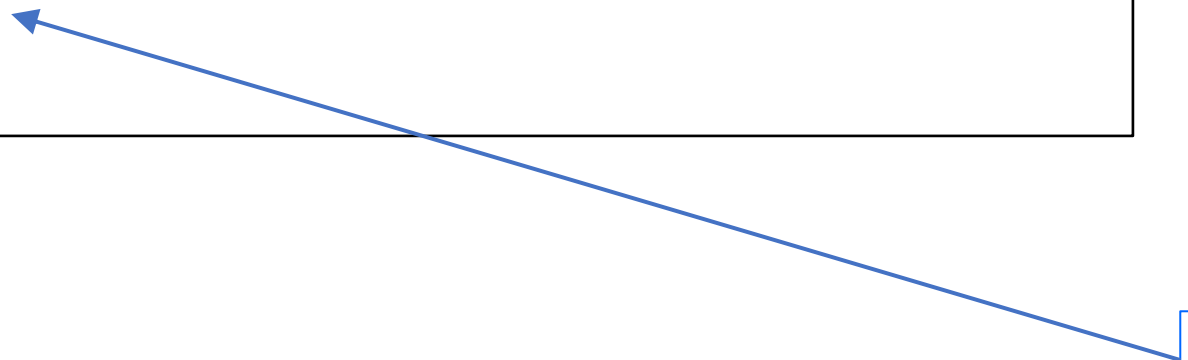
```
vector<string> names = {"Sarah", "William", "Alfredo"};
for( auto it = names.begin(); it!=names.end(); ++it) cout << *it;
```

- Not recommended when 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];
```

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

Plugging Your Own Functions into an Algorithm

- Many of the function templates in the STL are designed to accept function pointers as arguments.

- This allows you to “plug” one of your own functions into the algorithm.

Function pointers - review

- C++ doesn’t require that pointers point only to data;
- Function pointers point to memory addresses where functions are stored, e.g.

```
void (*fp) (void);
```

- A function’s name may be viewed as a constant function pointer.

Function pointers – review, cont.

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

void func_0() {
    cout << "I am func_0():" << endl;
}

int main() {
    void (*fp) () = func_0;
    fp();
    (*fp)();
    return 0;
}
```

```
I am func_0():
I am func_0():
```

- Function name may be viewed as a const pointer.
- Parameters and return type must match

- Either form may be used to invoke the function

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 – 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
```

STL Algorithms – search with 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
```

STL Algorithms – search with functor 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; }
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
```

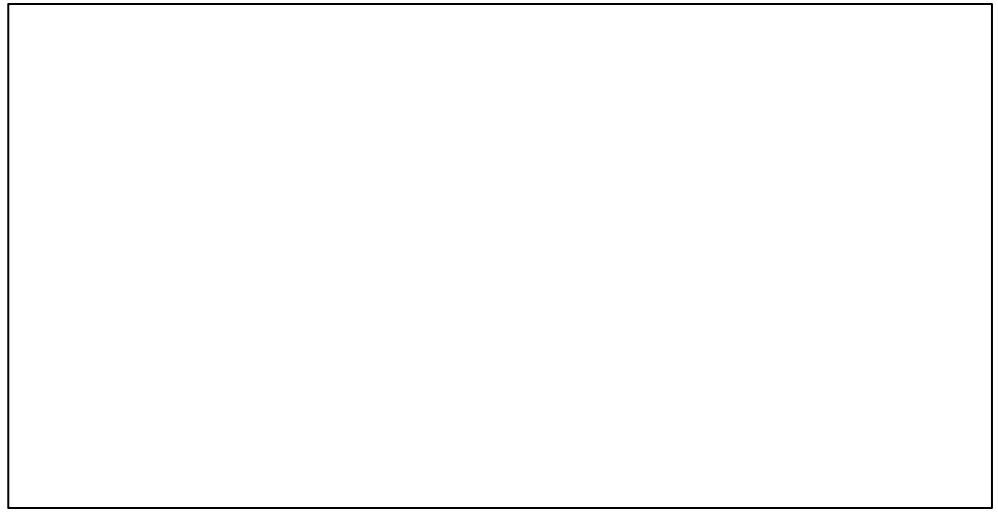
- **Functor**
  - Also known as “function object”.
  - An object whose class implements the function call operator.
- Anonymous Function Objects as predicates

Example 2

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

class Complex {
public:
    explicit Complex(double iReal = 0, double iImag = 0) : real(iReal), imag(iImag)
{}
    friend ostream& operator<<(ostream& os, const Complex rhs) {
        os << rhs.real << ((rhs.imag >= 0) ? "+" : "") << rhs.imag << 'i';
        return os;
    }
    friend bool operator==(const Complex& lhs, const Complex& rhs) {
        return lhs.real == rhs.real && lhs.imag == rhs.imag;
    }
    friend bool operator!=(const Complex& lhs, const Complex& rhs) {
        return !(lhs.real == rhs.real) || !(lhs.imag == rhs.imag);
    }
    Complex operator+(const Complex& rhs) const {
        Complex result;
        result.real = real + rhs.real;
        result.imag = imag + rhs.imag;
        return result;
    }
    Complex& operator+=(const Complex& rhs) {
        real += rhs.real;
        imag += rhs.imag;
        return *this;
    }
    Complex& operator++() {
        real++;
        return *this;
    }
    Complex operator++(int dummy) {
        Complex original(*this);
        real++;
        return original;
    }
    operator bool() const { return real || imag; }
    // Some getters
    double get_real() const { return real; }
    double get_imag() const { return imag; }
    double get_mag() const { return sqrt(real * real + imag * imag); }
private:
    double real;
    double imag;
};
```

Function pointers as predicates

```
bool BiggerThan5(const Complex& val) {
    return val.get_mag() > 5;
}


bool compare(Complex a, Complex b) {
    return a.get_mag() < b.get_mag();
}
```

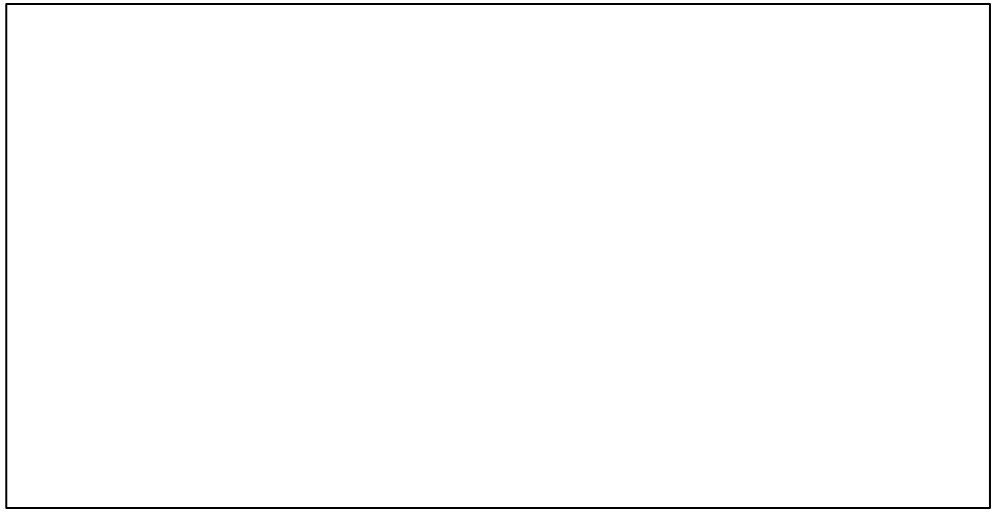


Function pointers as predicates - example

```
int main() {
    vector<Complex> myvec = { Complex(3,4), Complex(1,3),
    Complex(3,15), Complex(4,7), Complex(17,23),
    Complex(14,4), Complex(15,30), Complex(0,0) };

    // Display the contents of myvec
    cout << "Displaying vector contents: " << endl;
    for (const Complex& val : myvec) { cout << val << " "; }
    cout << endl;

    // use the find_if() with a function pointer
    cout << endl << "Demo find_if with function pointer: “
    << endl;
    cout << *find_if(myvec.begin(), myvec.end(),
    BiggerThan5);


}
```

Displaying vector contents:
3+4i 1+3i 3+15i 4+7i 17+23i 14+4i 15+30i 0+0i

Demo find_if with function pointer:
3+15i



Function objects as predicates

```
class CBiggerThan5 {
public:
    bool operator() (Complex val) { return val.get_mag() > 5; }
};


class Compare {
public:
    bool operator()(Complex a, Complex b) {
        return a.get_mag() < b.get_mag();
    }
};
```

Function objects as predicates

```
int main() {
    vector<Complex> myvec = { Complex(3,4), Complex(1,3), Complex(3,15),
Complex(4,7),
                              Complex(17,23), Complex(14,4), Complex(15,30),
Complex(0,0) };

    // Display the contents of myvec
    cout << "Displaying vector contents: " << endl;
    for (const Complex& val : myvec) { cout << val << " "; }
    cout << endl;

    // use the find_if() with a function object
    cout << endl << "Demo find_if with function object: " << endl;
    CBiggerThan5 cbigger;
    cout << *find_if(myvec.begin(), myvec.end(), cbigger) << endl;
    cout << *find_if(myvec.begin(), myvec.end(), CBiggerThan5()) << endl;


}
```

```
Displaying vector contents:
3+4i 1+3i 3+15i 4+7i 17+23i 14+4i 15+30i 0+0i

Demo find_if with function object:
3+15i
3+15i
```