# Functors and Algorithms

Rupesh Nasre.

January 2025
OOAIA

# Functors

- These are function objects.
- Called as class names.
- Implemented as parenthesis operator in C++.
  - void operator ()(*args*) {...}

# Function (not functor)

```
#include <bits/stdc++.h>
using namespace std;

int increment(int x) {  return (x+1); }

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(*arr);

    transform(arr, arr + n, arr, increment);

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    cout << endl;

    return 0;
}
```

**Transform is an algorithm. Increment is a function.**

**Can you increment by K?**

# Functor

```
#include <bits/stdc++.h>
using namespace std;

class increment {
public:
    int operator () (int arr_num) {
        return arr_num + 1;
    }
};

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);

    transform(arr, arr+n, arr, increment());

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}
```

Depending upon the g++ version, parentheses are optional.

**Increment is a functor.**

# Functor with Aggregates

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

struct vfun {
        int operator()(int n) {
                cout << n << endl;
                return n+1;
        }
};
int main() {
        vector<int> v;
        v.push_back(2);
        v.push_back(3);
        v.push_back(1);
        v.push_back(9);

        transform(v.begin(), v.end(), v.begin(), vfun());

        return 0;
}
```

**Classwork: Add k to each element of v.**

# Parameterized Functor

```
class increment {
public:
    increment(int ln) { n = ln; }
    int operator () (int arr_num) {
        return arr_num + n;
    }
private:    int n;
};
int main() {
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr) / sizeof(arr[0]);
    int k;

    cin >> k;
    transform(arr, arr+n, arr, increment(k));

    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}
```

**Can use constructor to store the argument.**
**Functors permit stateful update (over functions).**

```
class increment {
public:
    increment(int ln) { n = ln; }
    int operator () (int arr_num) {  return arr_num + n;  }
private: int n;
};
int main() {
    vector<int> arr;
        arr.push_back(1);       arr.push_back(2);       arr.push_back(3);
        arr.push_back(4);       arr.push_back(5);
    int n = arr.size();
    increment inc(20);

    transform(arr.begin(), arr.end(), arr.begin(), inc);

    // method 1: indexing
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
    cout << endl;

    // method 2: iterators
    for (vector<int>::iterator it = arr.begin(); it != arr.end(); ++it)
        cout << *it << " ";
    cout << endl;

    // method 3: algorithm
    copy(arr.begin(), arr.end(), std::ostream_iterator<int>(cout, " "));
}
```

# #include <algorithm>

- for_each
- find, find_if
- count, count_if, copy, copy_if
- equal, unique, reverse
- sort, is_sorted, lower_bound, binary_search
- transform
- Much more ...

**On a side note, <bits/stdc++> makes life easy, but is not part of the C++ standard (as of 2025)**

```
struct lessthanthree {
	bool operator()(int n) {  return n < 3;  }
};
struct tworaisedtoprint {
	void operator()(int n) {  cout << (1 << n) << " ";   }
};
struct tworaisedto {
	int operator()(int n) {  return (1 << n);   }
};
int main() {
	vector<int> v;
	v.push_back(1);	v.push_back(2);	v.push_back(3);
	v.push_back(4);	v.push_back(5);

	int small = count_if(v.begin(), v.end(), lessthanthree());
	cout << "Number of elements less than 3 = " << small << endl;

	for_each(v.begin(), v.end(), tworaisedtoprint());
	cout << endl;

	transform(v.begin(), v.end(), v.begin(), tworaisedto());
	reverse(v.begin(), v.end());
	for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
		cout << *it << " ";
	}
	cout << endl;
	return 0;
}
```

# Classwork

- Write main and functor to convert each string w in a list to ww.

# Unary and Binary Functors

- **transform**(in1.begin(), in1.end(),

  out.begin(), classname);
- **transform**(in1.begin(), in1.end(), in2.begin(), out.begin(), classname);

```
vector<string> v;
vector<string> v2;

v.push_back("One");        v.push_back("Two");
v.push_back("Three");      v.push_back("Four");
v2.push_back("1");         v2.push_back("2");
v2.push_back("3");         v2.push_back("4");

transform(v2.begin(), v2.end(), v2.begin(), dup());
transform(v.begin(), v.end(), v2.begin(), v2.begin(), cat());
```

# Lambda Expressions

- Functions and functors are named, hence reusable.
  - fun(x); fun(y); fun(z);
- When reusability is not warranted, name is also unnecessary.
- Basic syntax
  - Function: rettype name(params) { function definition }
  - Lambda: [capture] (params) -> rettype { function definition }

```
int main() {
        [ ](int x) {cout << x << endl;};
        return 0;
}
```

Defines the lambda, but does not call it.

# Lambda Expressions

```
int main() {
        vector<int> v = {0, 1, 2, 3};
        for_each(v.begin(), v.end(), [ ](int x) {cout << x << endl;});
        return 0;
}
```

**Write code to increment each vector element by 1.**

```
int main() {
        [ ](int x) {cout << x << endl;};
        return 0;
}
```

Defines the lambda, but does not call it.

# Lambda Expressions

```
int main() {
        vector<int> v = {0, 1, 2, 3};
        for_each(v.begin(), v.end(), [ ](int &x) {++x;});
        for_each(v.begin(), v.end(), [ ](int x) {cout << x << endl;});
        return 0;
}
```

**Write code to increment each vector element by 1.**

```
int main() {
        vector<int> v = {0, 1, 2, 3};
        int k = 3;
        for_each(v.begin(), v.end(), [ ](int &x) {++x;});
        for_each(v.begin(), v.end(), [ ](int x) {cout << x << endl;});
        return 0;
}
```

**Write code to increment each vector element by k.**

# Lambda Expressions

```
int main() {
        vector<int> v = {0, 1, 2, 3};
        int k = 3;
        for_each(v.begin(), v.end(), [ ](int &x) {x += k;});
        for_each(v.begin(), v.end(), [ ](int x) {cout << x << endl;});
        return 0;
}
```

**Write code to increment each vector element by k.**

```
$ g++ file.cpp
  error: ‘k’ is not captured
```

# Accessing Variables

- Lambdas have a different scope, and cannot access outside variables directly.
- However, since they are within a function / block scope, it is useful if they can access outside variables.
- This is done using the capture clause [capture].
- By default, no outside variables are captured.

```
int main() {
        vector<int> v = {0, 1, 2, 3};
        int k = 3;
        for_each(v.begin(), v.end(), [=](int &x) {x += k;});
        for_each(v.begin(), v.end(), [ ](int x) {cout << x << endl;});
        return 0;
}
```

Capture by value

Can also access v here.

**Note: Similar to functions, globals are by default captured.**

# Capture Clause

- [=] Capture all by value
- [&] Capture all by reference
- [a, &b, c, &d] Capture a and c by value; b and d by reference.

```
int g = 10;
int main() {
        std::vector<int> v = {0, 1, 2, 3};
        int k1 = 3, k2 = 5;
        for_each(v.begin(), v.end(), [&](int &x) {
                std::cout << ++k1 << ++k2 << ++g << std::endl;
        });
        for_each(v.begin(), v.end(), [ ](int x) {
                std::cout << x << std::endl;
        });
        return 0;
}
```

What is the output if capture is changed to =?

# Find the output.

```
std::vector<int> v(10);


std::iota(v.begin(), v.end(), 1);

for_each(v.begin(), v.end(), [ ](int x) {
    std::cout << x << ' ';
}); std::cout << std::endl;

int nodd = count_if(v.begin(), v.end(), [ ](int x) {
    return x % 2;
});
std::cout << "odd = " << nodd << std::endl;
```