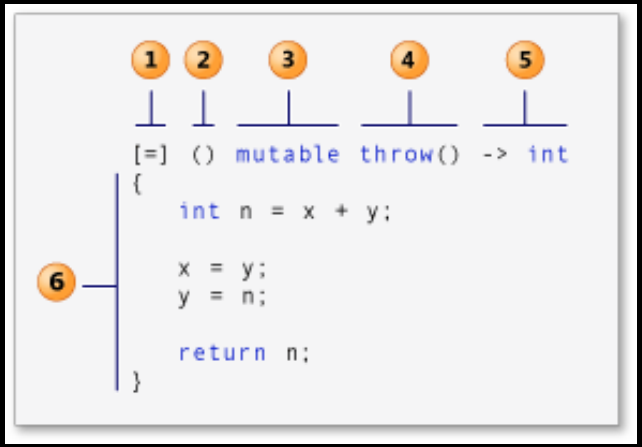
LAMBDA EXPRESSION 

.:: Fundamentals ::.



1. **CLÁUSULA DE CAPTURA** (también conocida como *iniciador de expresión lambda* en la especificación de C++)

2. **LISTA DE PARÁMETROS** Opcional (también conocida como *declarador de expresión lambda*)

3. **ESPECIFICACIÓN MUTABLE** Opcional

4. **ESPECIFICACIÓN DE EXCEPCIÓN** Opcional

5. **TIPO DE VALOR DEVUELTO FINAL** Opcional

6. **CUERPO DE LA EXPRESIÓN** *lambda*

```cpp
/* 1. Declarar expresiones lambda */

#include <functional>
#include <iostream>

using namespace std;

int main(){

    auto f1 = [](int x, int y) { return x + y; };
    cout << f1(2, 3) << endl;
    //
    function<int(int, int)> f2 = [](int x, int y) { return x + y; };
    cout << f2(3, 4) << endl;
    //
    auto cuadrado = [](int x) { return x * x; }(5);
    cout << cuadrado << endl;


}
```

```cpp
/* 2. Llamar a una expresion lambda */
#include <iostream>
using namespace std;

int main(){

    int sum = [] (int x, int y) { return x + y; }(5, 4);
    auto mul = [] (int x, int y) { return x * y; };

    cout <<"sum(5,4)=> "<< sum << endl;
    cout <<"mul(5,4)=> "<< mul(5,3) << endl;
}
```

```cpp
/* 3. Anida expresiones lambda */

#include <iostream>

using namespace std;

int main(){

    int duplicarDosVeces =
        [](int x) {
            return [](int y) { return y * 2;}(x)
            * 2;
        }(5);


    cout <<"duplicar dos veces (5) => "<< duplicarDosVeces << endl;

}
```

```cpp
/* 3. Anida expresiones lambda */

#include <iostream>

using namespace std;

int main(){

    int duplicarDosVeces =
        [](int x) {
            return ( [](int y) { return y * 2;}(x)
            * 2
                   );
        }(5);

    cout <<"duplicar dos veces (5) => "<< duplicarDosVeces << endl;

}
```

```cpp
// 4. Funciones Lambda de orden superior
// compile with: /EHsc /W4

#include <iostream>
#include <functional>

using namespace std;

int main(){

    auto addtwointegers =
        []
        (int x)
        -> function<int(int)>
        {
          return [=](int y) { return x + y; };
        };
    //-----
    auto higherorder =
        []
        (const function<int(int)>& f, int z)
        { return f(z) * 2; };

    //----------
    auto answer = higherorder(addtwointegers(7), 8);
    cout << answer << endl;
}
```

```cpp
// 5.0 Using a Lambda Expression in a Function

#include <algorithm>
#include <iostream>
#include <vector>

using namespace std;

class Scale{
private:
    int _scale;
public:
    explicit Scale(int scale) : _scale(scale) {}

    void ApplyScale(const vector<int>& v) const {
        for_each(
            v.begin(),
            v.end(),
            [=](int n) { cout << n * _scale << endl; }
        );
    }
};
///---
int main(){
    vector<int> values;
    values.push_back(1);
    values.push_back(2);
    values.push_back(3);
    values.push_back(4);

    Scale s(3);
    s.ApplyScale(values);

}
```

```cpp
// 6.0 template_lambda_expression.cpp

#include <vector>
#include <algorithm>
#include <iostream>

using namespace std;


template <typename T>
void negate_all(vector<T>& v){
    for_each(
        v.begin(),
        v.end(),
        [](T& n) { n = -n; }
    );
}

template <typename T>
void print_all(const vector<T>& v){
    for_each(
        v.begin(),
        v.end(),
        [](const T& n) { cout << n << endl;}
    );
}


int main(){

    vector<int> v;
    v.push_back(34);
    v.push_back(-43);
    v.push_back(56);

    print_all(v);
    negate_all(v);
    cout << "After negate_all():" << endl;
    print_all(v);
}
```