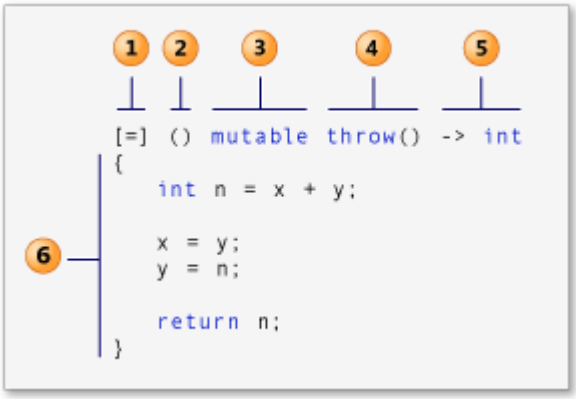
LAMBDA EXPRESSIONS (LAMBDA) IN C++

Lambda is a convenient way of defining an anonymous function object right at the location where it's invoked or passed as an argument to a function. Lambda is easy to read too because we can keep everything in the same place.

Lambda syntax is defined as:



1. ***capture clause*** (Also known as the *lambda-introducer* in the C++ specification.)
2. ***parameter list*** Optional. (Also known as the *lambda declarator*)
3. ***mutable specification*** Optional.
4. ***exception-specification*** Optional.
5. ***trailing-return-type*** Optional.
6. ***lambda body***.

1. Capture clause

A capture clause of lambda definition is used to specify which variables are captured and whether they are captured by reference or by value.
An empty capture closure **[ ]**, indicates that no variables are used by lambda which means it can only access variables that are local to it.
The capture-default mode indicates how to capture outside the variables referenced in Lambda:

- The capture closure **[&]** means the variables are captured by reference.
- The capture closure **[=]** indicates that the variables are captured by value.

✓ If a capture clause includes a capture-default **&**, then no identifier in a capture of that capture clause can have the form **&identifier**.
✓ Likewise, if the capture clause includes a capture-default **=**, then no capture of that capture clause can have the form **=identifier**.
✓ An **identifier** or **this** can't appear more than **once** in a capture clause.

The following code snippet illustrates some examples:

```
[&, i] {};                  // OK
[&, &i] {};                 // ERROR: i preceded by & when & is the default
[=, this] {};               // ERROR: this when = is the default
[=, *this] { };             // OK: captures this by value. See below.
[i, i] {};                  // ERROR: i repeated
[id, ft, &ch, &bl] {};          // OK: Mixing captures
```

Generalized Lambda expressions in C++14:

C++ 14 buffed up lambda expressions further by introducing what's called a *generalized* lambda. To understand this feature lets take a general example.

Suppose we create a lambda function to return the sum of two integers. So, our lambda function would look like below

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

int main(){
    auto func=[](int a, int b) -> int{
        return a+b;
    };

    cout<<"int: "<<func(10,20)<<endl;
    return 0;
}
```

Suppose we create a lambda function to return the sum of two floating point values. So, we would need to declare another lambda expression that would work for only double values.

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

int main(){
    auto func=[](int a, int b) -> int{
        return a+b;
    };

    auto func1=[](double a, double b) -> double{
        return a+b;
    };

    cout<<"Int: "<<func(10,20)<<endl;
    cout<<"Double: "<<func1(10.30,20.80)<<endl;
    return 0;
}
```

Before C++ 14 there was a way to circumvent this problem by using template parameters, C++11 lambdas can't be templated as stated but decltype() seems to help when using a lambda within a templated class or function.

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

template<typename T>
void sampleFun(T a, T b){
    auto func = [](decltype(a) a, decltype(b) b) -> T { return a + b; };
```

```
    cout<<"res: "<<func(a,b)<<endl;
}
int main(){
    sampleFun(10,20);
    sampleFun(10.30, 20.80);
    return 0;
}
```

C++ 14 does away with this and allows us to use the keyword *auto* in the input parameters of the lambda expression. The compilers can now deduce the type of parameters during compile time. So, in our previous example, a lambda expression that would work for both integer, floating-point values and strings would be

```
#include <iostream>
#include <string>

using namespace std;
int main()
{
    auto sum = [](auto a, auto b) {
        return a + b;
    };

    cout << sum(10, 20) << endl;
    cout << sum(10.30, 20.80) << endl;
    cout << sum(string("Uday"), string("Kishore")) << endl;

    return 0;
}
```

2. Parameters

The parameter list of a lambda is just like the parameter list of any other method: a set of parameters inside braces **()**. Input parameters in a lambda expression can be listed while defining lambda expressions but are not required.

3. Mutable: (Optional) Enables variables captured by a call by value to be modified.
Typically, a lambda's function call operator is constant-by-value. As per the below code snippet:

```
int a=10, b=20;
    auto sum = [=]() {
        a++; b++;
        return a + b;
    };
```

We see the below errors when we try to increment both variables (a and b) from the above code.

```
error: increment of read-only variable 'a'
```

```
error: increment of read-only variable 'b'
```

But use of the **mutable** keyword cancels this out. It doesn't produce mutable data members. The **mutable** specification enables the body of a lambda expression to modify variables that are captured by value.

```
int main()
{
    int a=10, b=20;
    auto sum = [=]()mutable {
        a++; b++;
        return a + b;
    };

    cout << sum() << endl;
    return 0;
}
```

4. **Exception**: (Optional) Use "noexcept" to indicate that lambda does not throw an exception.

```
int main()
{
    []() noexcept { throw 5; }();
    return 0;
}
```

5. **Return type**: (Optional) The compiler deduces the return type of the expression on its own. But as lambdas get more complex, it is better to include return type as the compiler may not be able to deduce the return type.

```
auto x1 = [](int i){ return i; }; // OK: return type is int
auto x2 = []{ return{ 1, 2 }; };  // ERROR: return type is void, deducing
                                  // return type from braced-init-list isn't valid
```

6. **Method definition**: Lambda body.

The body of a C++ lambda expression is composed of all those things that the body of a normal function or a method is made up of. These are the type of variables that the body of a function or a lambda expression can hold:

- Local variables
- Global Variables
- Captured variables (variables within [])
- Arguments/Parameters
- Data members of a class