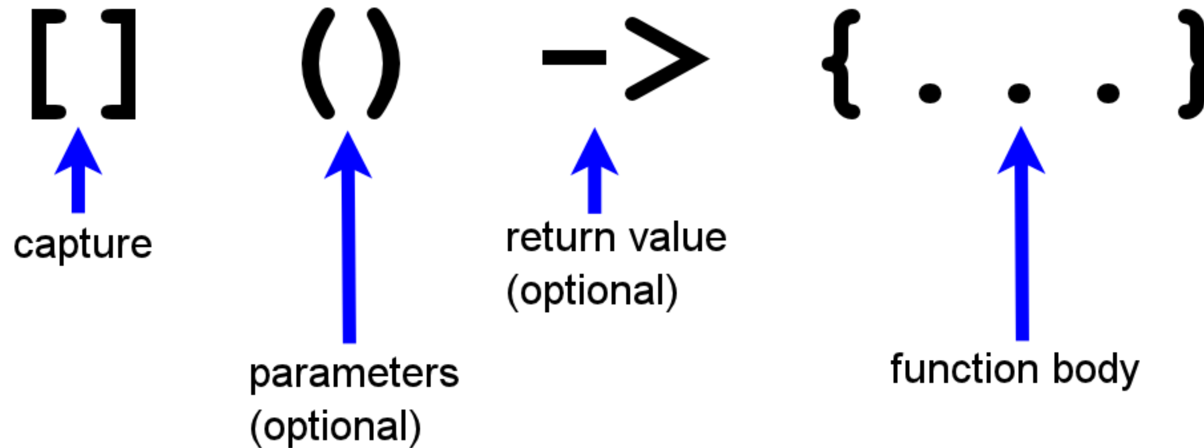
Lambda Functions

- Lambda functions
  - Are functions without a name.
  - Define their functionality just in place.
  - Could be copied like data.
  - Are able to store the invocation environment.
- Lambda functions should be
  - Short and concise.
  - Self-explanatory.

Lambda Functions: Syntax



- `[]`: Captures the used variables
- `()`: Necessary for parameters
- `->`: Necessary for complex lambda functions
- `{}`: Function body, per default `const`

  `[]() mutable -> {...}` has a not constant function body.

Lambda Functions: Closures

Lambda functions can bind their invocation context.

➡ Closure

Binding	Description
`[]`	no binding
`[a]`	a per copy
`[&a]`	a per reference
`[=]`	all used variables per copy
`[&]`	all used variables per reference
`[=,&a]`	per default per copy; a per reference
`[&,a]`	per default per reference; a per copy
`[this]`	data and member of the enclosing class per copy
`[l= std::move(lock)]`	moves lock (C++14)

Lambda Functions: First Class Function

Lambda functions can be stored in variables and can be returned from functions.

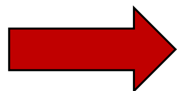
- First class function

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

std::thread th( []{std::cout <<  "Hello from thread" << std::endl;} );

std::function<int(int,int)> makeAdd(){
  return [](int a, int b){ return a + b; };
}

std::function<int(int, int)> myAdd= makeAdd();
myAdd(2000, 11);   // 2011
```

Lambda Functions: Implementation

A lambda function is a function object, that is instantiated just in place.

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



```
class TmpAdd{
public:
  int operator()(int a, int b) const {
    return a + b;
  }
};
TmpAdd tmpAdd;
```

```
add(2000, 11); // 2011
```

```
tmpAdd(2000, 11); // 2011
```

Generic Lambda Functions: C++14

In C++14 lambda functions can deduce their argument type.

```
auto add11=[ ](int i, int i2){ return i + i2; };
auto add14= [ ](auto i, auto i2){ return i + i2; };

std::vector<int> myVec{1, 2, 3, 4, 5};
auto res11= std::accumulate(myVec.begin(), myVec.end(), 0, add11);
auto res14= std::accumulate(myVec.begin(), myVec.end(), 0, add14);
  // res11 == res14 == 15;

std::vector<std::string> myVecStr{"Hello"s, " World"s};
auto st= std::accumulate(myVecStr.begin(), myVecStr.end(), ""s,
add14);
std::cout << st << std::endl;   // Hello World
```

Lambda Functions

- Examples :
  - `lambdaFunction.cpp`
  - `lambdaFunctionClosure.cpp`
  - `lambdaFunctionCapture.cpp`
  - `lambdaFunctionThis.cpp`
  - `lambdaFunctionGeneric.cpp`
- Exercises:
  - The program `lambdaFunctionCapture.cpp` has undefined behaviour. Fix it.
    - Solution: `lambdaFunctionCapture.cpp`
  - The rules for lambda functions become relatively fast difficult . Convince yourself.
- Further informations:
  - [lambda functions]()

Type-Traits

Enables to make type checks, type comparisons and type modfications at compile time

➡ No impact on the runtime

- Needs the header `<type_traits>`.

- Application of template metaprogramming
  - Programming at compile time
  - Programming with types and not with values
  - Compiler instantiates the templates and generates C++ code

Type-Traits: Goals

- Optimisation
  - Code which optimises itself
    - ➡ Depending on the type of a variable another code will be chosen
  - Optimised version of `std::copy`, `std::fill`, or `std::equal` is used
    - ➡ Algorithms can work on memory blocks
- Correctness
  - Type checks will be performed at compile time
  - Type informations define together with `static_assert` requirements for the code

Type-Traits

- Type checks
  - Primary type categories (`::value`)

    `std::is_pointer<T>, std::is_integral<T>, std::is_floating_point<T>`
  - Composed type categories (`::value`)

    `std::is_arithmetic<T>, std::is_object<T>`

- Type comparison (`::value`)

  `std::is_same<T, U>, std::is_base_of<Base, Derived>, std::is_convertible<From, To>`

- Type transformation (`::type`)

  `std::add_const<T>, std::remove_reference<T>, std::make_signed<T>, std::add_pointer<T>`

Type-Traits

- Examples:
  - `typeTraitsTypeCategories.cpp`
  - `typeTraitsCopy.cpp`
  - `typeTraitsGcd.cpp`
- Exercises:
  - Modify an `int` type at compile time.
    - Add `const` to your type.
    - Remove `const` from your type.
    - Compare your type with `const int`.
    - Solution: `typeModifications.cpp`
- Further information:
  - [Type-Traits]
  - [Further variations] of the gcd algorithm

Constant Expressions: constexpr

Constant Expressions

- Can be evaluated at compile time
- Give the compiler deep insight
- Are implicit thread-safe

- Variables `constexpr double myDouble= 5.2;`
  - Are implicit `const`.
- Functions

```
constexpr int fac(int n){ return n > 0 ?
                                     n * fac(n-1):
                                     1;
                          }
```

  - Have to return a value
  - Will be executed at compile time if invoked within a constant expression
  - Can only have a function body consisting out of a return statement
  - Must have a constant return value
  - Are implicitly `inline`

Constant Expressions: constexpr

- User-defined types

```
struct MyDouble{
   double myVal;
   constexpr MyDouble(double v): myVal(v){}
   constexpr double getVal(){return myVal;}
 };
```

  - The constructor has to be empty and a constant expression
  - The user-defined type can have methods which are constant expressions
  - Instances of `MyDouble` can be instantiated at compile time

- Functions with C++14
  - Can have variables that have to be initialised by a constant expression
  - Can have loops
  - Can have no static or `thread_local` variable

Constant Expressions: constexpr

- Examples:
  - `constExpression.cpp`
  - `constExpressionCpp14.cpp`
- Exercises:
  - Use `MyDouble` in your program.
    - How can you check that instances of `MyDouble` will be created at compile time?
    - What will happen if `MyDouble` is used in a non-constant expression?

```
struct MyDouble{
    double myVal;
    constexpr MyDouble(double v): myVal(v){}
    constexpr double getVal(){return myVal;}
  };
```

- Further information:
  - constexpr
  - Calculation of distances at compile time

Associative Containers

Associative Container	Keys sorted	Value available	Multiple identical keys	Access time	Standard
`std::set`	yes	no	no	logarithmic	C++98
`std::unordered_set`	no	no	no	constant	C++11
`std::map`	yes	yes	no	logarithmic	C++98
`std::unordered_map`	no	yes	no	constant	C++11
`std::multiset`	yes	no	yes	logarithmic	C++98
`std::unordered_multiset`	no	no	yes	constant	C++11
`std::multimap`	yes	yes	yes	logarithmic	C++98
`std::unordered_multimap`	no	yes	yes	constant	C++11