STL Functors

Contents

[1. Introduction 2](#_Toc100061597)

[What are functors? 2](#_Toc100061598)

[Example: 2](#_Toc100061599)

[2. Functors 2](#_Toc100061600)

[Why do we need functors? 3](#_Toc100061601)

[Separation of concerns 3](#_Toc100061602)

[Parameterization 4](#_Toc100061603)

[Statefulness 4](#_Toc100061604)

[Performance 4](#_Toc100061605)

[3. STL Functors 5](#_Toc100061606)

[Arithmetic Operations 5](#_Toc100061607)

[std::plus 5](#_Toc100061608)

[std::minus 6](#_Toc100061609)

[std::multiplies 6](#_Toc100061610)

[std::divides 7](#_Toc100061611)

[std::modulus 7](#_Toc100061612)

[std::negate 7](#_Toc100061613)

[Comparison Operations 8](#_Toc100061614)

[Logical Operations 9](#_Toc100061615)

[Bitwise Operations 9](#_Toc100061616)

# Introduction

## What are functors?

* Functors are **objects (note that these are objects)** that can be treated as though they are a **function** or **function pointer**.
* Let’s understand why we need functors by example...

## Example:

* Let’s say we want to increment each element in a *vector*. So, we can write…

|  |
| --- |
| int increment(int x) { return (x + 1); }  int main() {  *vector*<int> arr = { 1, 2, 3, 4, 5 };  // 'transform' Applies an operation sequentially to the  // elements arr and stores the result in arr.  *transform*(arr.*begin*(), arr.*end*(), arr.*begin*(), increment);    for(auto i: arr)  *cout* << i << " "; // 2 3 4 5 6  return 0;  } |

* Let’s say now the requirement is to add 6 to every element in the *vector* and store the result back in *vector*.
* As *transform* requires a unary function (a function taking only one argument), we cannot pass a number to increment().
* So, we must write several different functions to add each number!!!
* Functors can solve the above problem…

# Functors

* A functor (or function object) is a C++ class that acts like a function.
* Functors are called using the same old function call syntax.
* To create a functor, we create an object that overloads the operator().

|  |
| --- |
| class increment {  private:  int num;  public:  increment(int n) : num(n) { }  // Overloaded operator!!!  int operator () (int arr\_num) const {  return num + arr\_num;  }  };  int main() {  *vector*<int> arr = { 1, 2, 3, 4, 5 };  int numToAdd = 0;  *cout* << "Enter the number to be added: ";  *cin* >> numToAdd; // 6    // 'transform' Applies an operation sequentially to the  // elements arr and stores the result in arr.  *transform*(arr.*begin*(), arr.*end*(), arr.*begin*(), increment(numToAdd));    for(auto i: arr)  *cout* << i << " "; // 7 8 9 10 11  return 0;  } |

## Why do we need functors?

* Benefits of using a functor over a function…
  1. Separation of concerns.
  2. Parameterization.
  3. Statefulness.
  4. Performance.

### Separation of concerns

|  |
| --- |
| class CalculateAverage {  private:  *std*::*size\_t* num;  double sum;  public:  CalculateAverage() : num(0), sum(0) { }  void operator () (double elem) {  num++;  sum += elem;  }  operator double() const {  return sum / num;  }  };  int main() {  *vector*<int> arr = { 1, 2, 3, 4, 5 };  double average = *std*::*for\_each*(arr.*begin*(), arr.*end*(), CalculateAverage());  *cout* << average << "\n"; // 3  return 0;  } |

* In the above example, the functor-based approach has the advantage of separating the iteration logic from the average-calculation logic.

### Parameterization

* As we saw, we can parameterize… We could of course do the same thing with a traditional function, but then makes it difficult to use with function pointer.

|  |
| --- |
| class increment {  private:  int num;  public:  increment(int n) : num(n) { }  // Overloaded operator!!!  int operator () (int arr\_num) const {  return num + arr\_num;  }  }; |

### Statefulness

* Unlike Functions Functor can have state.

|  |
| --- |
| class Matcher {  int target; // Holds the state!  public:  Matcher(int m) : target(m) {}  bool operator()(int x) { return x == target; }  };  int main() {  Matcher is10(10);  *cout* << "Enter a value: ";  int ele; *cin* >> ele;  if (is10(ele))  *cout* << "Entered value is equal to 10\n";  else  *cout* << "Entered value is not equal to 10\n";  return 0;  } |

### Performance

* Functors can often be in-lined by the compiler. Whilst the same is theoretically true of functions, compilers typically won't inline through a function pointer.

# STL Functors

* STL includes a set of **template classes** that overload the function call operator (operator ()).
* **Instances** of those classes are called **functors** or function objects.
* STL has two kinds of function objects:
  1. Unary Functor: Functor that can be called with one argument.
  2. Binary Functor: Functor that can be called with two arguments.
* A predicate is a specific kind of functor: a functor that evaluates to a boolean value.
* Among STL functors there is a **group of function objects** called **predicate** which take one or two arguments and **return** bool**ean value** or object convertible to boolean value.
  1. The predicates which take one argument are called unary predicates.
  2. Those who take two arguments are called binary predicates.
* STL functors are declared in the header <functional> and are part of namespace std.
* They are divided in following groups according to their functionality:
  1. Functors for **Arithmetic Operations**.
  2. Functors for **Comparison Operations**.
  3. Functors for **Logical Operations**.
  4. Functors for **Bitwise Operations**.

## Arithmetic Operations

* They are called for arithmetic operations like addition, subtraction, etc.
* STL provides following arithmetic functors…
  + std::plus
  + std::minus
  + std::multiplies
  + std::divides
  + std::modulus
  + std::negate

### std::plus

* *std::plus* is a binary functor which take two operands and call the operator + for them.
* The default template argument is void and it is specialized for void type where its function operator deduce the argument type and return type from the arguments.

|  |
| --- |
| int main() {  *std*::*string* s1 = "Hello ";  const char\* s2 = "World";  *std*::*plus*<*std*::*string*> stringAdder3;// Adds two string objects.  *std*::*plus*<> stringAdder1;// default type is void,template specialization used.  *std*::*plus*<void> stringAdder2;// template specialization used.  *std*::*cout* << stringAdder1(s1, s2).*c\_str*() << '\n'; // Hello World  *std*::*cout* << stringAdder2(s1, s2).*c\_str*() << '\n'; // Hello World  *std*::*cout* << stringAdder3(s1, s2).*c\_str*() << '\n'; // Hello World  int a = 5;  int b = 5;  *std*::*cout* << "a+b: " << *std*::*plus*<int>{}(a, b) << '\n'; // 10  *vector*<int> v1 = { 10, 20, 30, 40, 50 };  *vector*<int> v2 = { 11, 21, 31, 41, 51 };  *vector*<int> r(5);  *std*::*transform*(v1.*begin*(), v1.*end*(), v2.*begin*(), r.*begin*(), *std*::*plus*<int>());  for (auto i : r) *cout* << i << " "; // 21 41 61 81 101  *cout* << '\n';  return 0;  } |

### std::minus

* *std::minus* is a binary functor which takes two operands and calls the operator - for them.

|  |
| --- |
| int main() {  *std*::*minus*<int> intsubtructor;  *cout* << "2000 - 1500: " << intsubtructor(2000, 1500) << '\n'; // 500  *vector*<int> v1 = { 10, 20, 30, 40, 50 };  *vector*<int> v2 = { 11, 21, 31, 41, 51 };  *vector*<int> r(5);  *std*::*transform*(v1.*begin*(), v1.*end*(), v2.*begin*(), r.*begin*(), *std*::*minus*<int>());  for (auto i : r) *cout* << i << " "; // -1 -1 -1 -1 -1  *cout* << '\n';  } |

### std::multiplies

* *std::multiplies* is a binary functor which take two operands and call the operator \* for them.

|  |
| --- |
| int main() {  *std*::*multiplies*<int> intMultiplier;  *cout* << "16 \* 15: " << intMultiplier(16, 15) << '\n'; // 240  *vector*<int> v1 = { 10, 20, 30, 40, 50 };  *vector*<int> v2 = { 11, 21, 31, 41, 51 };  *vector*<int> r(5);  *std*::*transform*(v1.*begin*(), v1.*end*(), v2.*begin*(), r.*begin*(), *std*::*multiplies*<int>());  for (auto i : r) *cout* << i << " "; // 110 420 930 1640 2550  *cout* << '\n';  return 0;  } |

### std::divides

* *std::divides* is a binary functor which take two operands and call the operator / for them.

|  |
| --- |
| int main() {  *cout* << "10 / 2: " << *std*::*divides*<int>()(10, 2) << '\n'; // 5  *vector*<float> v1 = { 11, 21, 31, 41, 51 };  *vector*<float> v2 = { 10, 20, 30, 40, 50 };  *vector*<float> r(5);  *std*::*transform*(v1.*begin*(), v1.*end*(), v2.*begin*(), r.*begin*(),  *std*::*divides*<float>());  for (auto i : r) *cout* << i << " "; // 1.1 1.05 1.03333 1.025 1.02  *cout* << '\n';  return 0;  } |

### std::modulus

* *std::modulus* is a binary functor which take two operands and call the operator % for them.

|  |
| --- |
| int main() {  *cout* << "10 % 2: " << *std*::*modulus*<int>()(10, 2) << '\n'; // 0  *vector*<int> v1 = { 11, 21, 31, 41, 51 };  *vector*<int> v2 = { 10, 20, 30, 40, 50 };  *vector*<int> r(5);  *std*::*transform*(v1.*begin*(), v1.*end*(), v2.*begin*(), r.*begin*(),  *std*::*modulus*<int>());  for (auto i : r) *cout* << i << " "; // 1 1 1 1 1  *cout* << '\n';  return 0;  } |

### std::negate

* *std::negate* is a unary functor which take one operand and call the operator - for the argument of type.

|  |
| --- |
| int main() {  *cout* << "INT\_MAX: " << *INT\_MAX* << '\n'; // 2147483647  *cout* << "Minus of INT\_MAX: " <<  *std*::*negate*<int>()(*INT\_MAX*) << '\n'; // -2147483647  *vector*<int> v1 = { 11, 21, 31, 41, 51 };  *transform*(v1.*begin*(), v1.*end*(), v1.*begin*(),  *std*::*negate*<int>());  for (auto i : v1) *cout* << i << " "; // -11 -21 -31 -41 -51  *cout* << '\n';  return 0;  } |

## Comparison Operations

* They are called for comparing two values like equality or inequality.
* STL provides following comparison functors…
  + *std::<greater>*: This is a binary functor which takes two operands and call the operator > for the arguments of type T.
  + *std::equal\_to*: This is a binary functor which takes two operands and call the operator == for the arguments of type T.
  + *std::not\_equal\_to*: This is a binary functor which takes two operands and call the operator != for the arguments of type T.
  + *std::less*: Checks if the first argument is less than the second argument.
  + *std::less\_equal*: Checks if the first argument is less than or equal to the second argument.
  + *std:: greater\_equal*: Checks if the first argument is greater than or equal to the second argument.

|  |
| --- |
| int main() {  int a = 10, b = 5;  *cout* << "a<b? " << *std*::*boolalpha* <<  *std*::*less*<int>()(a, b) << "\n"; // False  *cout* << "a>b? " << *std*::*boolalpha* <<  *std*::*greater*<int>()(a, b) << "\n"; // True  *cout* << "a==b? " << *std*::*boolalpha* <<  *std*::*equal\_to*<int>()(a, b) << "\n"; // False  *cout* << "a!=b? " << *std*::*boolalpha* <<  *std*::*not\_equal\_to*<int>()(a, b) << "\n"; // True  *cout* << "a>=b? " << *std*::*boolalpha* <<  *std*::*greater\_equal*<int>()(a, b) << "\n"; // True  *cout* << "a<=b? " << *std*::*boolalpha* <<  *std*::*less\_equal*<int>()(a, b) << "\n"; // False  *vector*<int> v1 = { 21, 11, 41, 31, 51 };    *sort*(v1.*begin*(), v1.*end*(), *std*::*less*<int>());  for (auto i : v1) *cout* << i << " "; // 11 21 31 41 51  *cout* << '\n';  *sort*(v1.*begin*(), v1.*end*(), *std*::*greater*<int>());  for (auto i : v1) *cout* << i << " "; // 51 41 31 21 11  *cout* << '\n';  return 0;  } |

## Logical Operations

* They are called for logical operation like logical AND, OR, etc.
* Logical operation functors *std::logical\_and* and *std::logical\_or* are binary functors which call operators *&&* and *||* on the arguments.
* *std::logical\_not* is a unary functor which calls the *!* Operator on its argument.

|  |
| --- |
| int main() {  int a = 10, b = 5;  *cout* << "a && b? " << *std*::*boolalpha* <<  *std*::*logical\_and*<int>()(a, b) << "\n"; // True  *cout* << "a || b? " << *std*::*boolalpha* <<  *std*::*logical\_or*<int>()(a, b) << "\n"; // True  *cout* << "!a is: " << *std*::*boolalpha* <<  *std*::*logical\_not* <int>()(a) << "\n"; // False  *vector*<int> v1 = { 11, 21, 31, 41, 0 };  *vector*<int> v2 = { 10, 20, 0, 40, 50 };  *vector*<bool> r(5);  *std*::*transform*(v1.*begin*(), v1.*end*(), v2.*begin*(), r.*begin*(),  *std*::*logical\_and*<int>());  for (auto i : r)  *cout* << *std*::*boolalpha* << i << " "; // true true false true false  *cout* << '\n';  return 0;  } |

## Bitwise Operations

* They are called to perform bitwise operations like ‘bitwise AND’, ‘bitwise OR’, etc.
* There are four functors for bit-wise operations:
  + *std::bit\_and* – Binary functor which can perform bit-wise *AND* operation, *x & y*.
  + *std::bit*\_or – Binary functor which can perform bit-wise *OR* operation, *x | y*.
  + *std::bit\_xor* – Binary functor which can perform bitwise *XOR* operation, *x ^ y*.
  + *std::bit\_not* – Unary functor which can perform bit wise *NOT* operation, *~x*.

|  |
| --- |
| int main() {  int a = 10, b = 5;  *cout* << "a & b: "<<*bitset*<sizeof(int)>(*bit\_and*<int>()(a, b)) << "\n";// 0000  *cout* << "a | b: "<<*bitset*<sizeof(int)>(*bit\_or*<int>()(a, b)) << "\n";// 1111  *cout* << "a ^ b: "<<*bitset*<sizeof(int)>(*bit\_xor* <int>()(a, b)) << "\n";// 1111  *cout* << "~a: " <<*bitset*<sizeof(int)>(*bit\_not*<int>()(a)) << "\n"; // 0101  return 0;  } |