MODERN C++ DESIGN PATTERNS

Functions

by Prasanna Ghali

Plan for Today

- Different things can be used as functions in C++
- Creating generic function objects
- What `std::bind<>` is and when to use it
- What lambdas are, and how they relate to ordinary function objects
- Creating prettier function objects
- What `std::function<>` is and when to use it

Lambdas

- So far, functions passed to algorithms already exist outside function you're using algorithms in
- Writing a proper function or whole class is tedious and possibly sign of bad software design
- Lambdas solve this problem
  - Syntactic sugar for creating unnamed function objects
  - Allow you to create function objects inline – at the place where you want them – instead of outside function you're currently writing
  - See *lambda0.cpp*

Lambda: Basic Syntax

- Syntactically, lambda expressions have 3 main parts: a head, an argument, and the body

Arguments

```
[a, &b] (int x, int y) { return a*x + b*y; }
```

Head

Optionals

Body

```
mutable
constexpr
exception attr
-> return type
```

Lambdas: Basic Syntax


```
std::vector<int> v {1, 3, 2, 5, 4};

// look for 3 ...
int three = 3;
int num_threes = std::count(v.begin(), v.end(), three);
// num_threes is 1

// look for values larger than three
auto is_above_3 = [](int v) { return v > 3; };
int num_above_3 = std::count_if(std::begin(v), std::end(v),
                                is_above_3);
std::cout << "num_above_3: " << num_above_3 << "\n";
```

Lambdas: Basic Syntax


```
std::vector<int> v {1, 3, 2, 5, 4};

// look for 3 ...
int three = 3;
int num_threes = std::count(v.begin(), v.end(), three);
// num_threes is 1

// look for values larger than three
int num_above_3 = std::count_if(std::begin(v), std::end(v),
                                [](int x) { return x > 3; });
std::cout << "num_above_3: " << num_above_3 << "\n";
```

stateless lambdas

Lambda Syntax: Head

- Specifies which variables from surrounding scope will be visible inside lambda body
- Variables can be captured as values or by references

Arguments

```
[a, &b] (int x, int y) { return a*x + b*y; }
```

Head

Body

Lambda Syntax: Head

- `[a, &b]` – a is captured by value; b by reference
- `[]` – nothing from outer scope is used
- `[&]` – outer scope variables are passed by reference
- `[=]` – outer scope variables are passed by value
- `[this]` – capture `this` pointer by value
- `[&, a]` – outer scope variables are passed by reference, except a, which is captured by value
- `[=, &b]` – outer scope variables are passed by value, except b, which is passed by reference

Lambdas: Capture Clause


```
int count_value_above(std::vector<int> const& v, int x) {
  auto is_above = [x](int i) { return i > x; };
  return std::count_if(std::begin(v), std::end(v),
                       is_above);
}
```

```
int count_value_above(std::vector<int> const& v, int x) {
  auto is_above = [&x](int i) { return i > x; };
  return std::count_if(v.begin(), v.end(), is_above);
}
```

Capture by Value Versus Capture by Reference


```cpp
std::vector<int> vi{1,2,3,4,5,6};
int x = 3;
auto is_above = [x](int v) {
  return v > x;
};
x = 4;
int count_b = std::count_if(
    std::begin(vi),
    std::end(vi),
    is_above
    );  // count_b is what value?
```

```cpp
std::vector<int> vi{1,2,3,4,5,6};
int x = 3;
auto is_above = [&x](int v) {
  return v > x;
};
x = 4;
int count_b = std::count_if(
    std::begin(vi),
    std::end(vi),
    is_above
    );  // count_b is what value?
```

Lambdas: Under the Hood [Capture by Value]


```cpp
int x {3};

auto is_above = [x](int y) {
  return y > x;
};

bool test = is_above(5);
```

```cpp
int x {3};

class IsAbove {
public:
  IsAbove(int vx) : x{vx} {}
  auto operator()(int y) const {
    return y > x;
  }
private:
  int x{}; // Value
};

IsAbove is_above{x};
bool test = is_above(5);
```

Lambdas: Under the Hood [Capture by Reference]


```cpp
int x {3};

auto is_above = [&x](int y) {
  return y > x;
};

bool test = is_above(5);
```

```cpp
int x {3};

class IsAbove {
public:
  IsAbove(int& rx) : x{rx} {}
  auto operator()(int y) const {
    return y > x;
  }
private:
  int &x; // Value
};

IsAbove is_above{x};
bool test = is_above(5);
```

Initializing Variables in Capture


```
auto some_func =
  [numbers = std::list<int>{4,2}]() {
  for (int i : numbers) {
    std::cout << i;
  }
};

some_func();  // output: 42
```

Initializing Variables in Capture


```
auto some_func =
  [numbers = std::list<int>{4,2}]() {
  for (int i : numbers) {
    std::cout << i;
  }
};

some_func(); // output: 42
```

```
class SomeFunc {
public:
  SomeFunc() : numbers{4, 2} {}
  void operator()() const {
    for (int i : numbers) {
      std::cout << i;
    }
  }
private:
  std::list<int> numbers;
};

SomeFunc some_func{};
some_func(); // Output: 42
```

Initializing Variables in Capture


```
int x {1};
auto some_func = [&y = x]() {
  // y is a reference to x
};
```

```
std::unique_ptr<int> x {std::make_unique<int>()};
auto some_func = [y = std::move(x)]() {
  // use y here..
};
```

Mutating Lambda Variables


```cpp
auto counter =  [count=10] () mutable {
  return ++count;
};

for (size_t i{}; i < 5; ++i) {
  std::cout << counter() << " ";
}
std::cout << "\n";
```

Mutating Lambda Variables


```cpp
int v {7};
auto lambda = [v]() mutable {
  std::cout << v << " ";
  ++v;
};
assert(v == 7);
lambda(); lambda();
assert(v == 7);
std::cout << v;
```

```cpp
class Lambda {
 public:
 Lambda(int m) : v{m} {}
 void operator()() {
   std::cout<< v << " ";
   ++v;
 }
private:
  int v{};
};
```

Mutating Lambda Variables


```cpp
int v {7};
auto lambda = [&v]() {
  std::cout << v << " ";
  ++v;
};
assert(v == 7);
lambda();
lambda();
assert(v == 9);
std::cout << v;
```

```cpp
class Lambda {
public:
  Lambda(int& m) : v{m} {}
  auto operator()() const {
    std::cout<< v << " "; ++v;
  }
private:
  int& v;
};
```

Capture All (1/4)


```
class Foo {
public:
  void member_function() {
    int a {0};
    float b {1.0f};
    // capture all variables by copy
    auto lambda0 = [=]() {std::cout << a << b;};
    // capture all variables by reference
    auto lambda1 = [&]() {std::cout << ++a << --b;};
 }
private:
  int m {};
};
```

Capture All (2/4)

- Using `[=]` or `[&]` doesn't mean all variables in scope are copied into lambda
  - Only variables actually used inside lambda are copied

Capture All (3/4)

- When capturing all variables by value [reference], you can specify variables to be captured by reference [value]

```
// assume variables a, b, c are in outer scope
// capture a, b, c by value
auto l1 = [=] { /* ... */ ; };
// capture a, b, c by reference
auto l2 = [&] { /* ... */ ; };
// capture a and b by value and c by reference
auto l3 = [=, &c] { /* ... */ ; };
// capture a and b by reference and c by value
auto l4 = [&, =c] { /* ... */ ; };
```

Capture All (4/4)

- Although convenient to capture all variables with [=] or [&], never a good idea
  - Performance penalties
  - Easier to interpret when specific identifiers labeled in capture list

Capturing `this` Pointer

- Member variables cannot be captured!!!

```
class Foo {
public:
  void member_function() {
    int a {0};
    float b {1.0f};
    auto lambda = [this]() {std::cout << ++m ;};
    // capture object by copy
    auto lambda2 = [*this]() {std::cout << m;};
    auto lambda3 = [=, this]() {std::cout << ++m ;};
    auto lambda4 = [a, &b, this]() {std::cout << m;};
 }
private:
  int m {};
};
```

Capture-Less Lambdas and Function Pointers

- Lambdas without capture can be implicitly converted to function pointers

```
extern void press_button(char const *msg,
     void (*callback)(int, char const*));

// + indicates lambda has no captures
auto lambda = +[](int result, char const *str) {
  // process result and str
};

press_button("pressed", lambda);
```

What About Lambdas With Capture?

- Lambdas with captures have their own unique type
- Even if two lambdas with capture are plain clones of each other, they still have their own unique type

Need for `std::function<>` (1/4)

- Function template `for_upto` can be used used with any callable [lambda, function pointer, function object]

```
template <typename F>
void for_upto(int N, F f) {
  for (int i{0}; i <= N; ++i) {
    f(i);
  }
}
```

Need for std::function<> (2/4)

- Relies on automatic type deduction to take lambda or function pointer ...

```
template <typename T>
void print(T t) { std::cout << t << ' '; }

// insert values from 0 to 4
std::vector<int> values;
for_upto(5, [&values](int i) { values.push_back(i); });

// print elements
for_upto(5, print<int>); printf("\n");
```

Need for `std::function<>` (3/4)

- Each use of `for_upto` will likely produce different instantiation
  - If `for_upto` was large, possible for its instantiations to increase code size
- One approach to limit increase in code size is to turn function template into non-template

```
void for_upto(int N, void (*f)(int)) {
  for (int i{0}; i != N; ++i) {
    f(i);
  }
}
```

Need for `std::function<>` (4/4)

- However, it'll produce an error when passed a lambda with capture

```
void for_upto(int N, void (*f)(int)) {
  for (int i{0}; i != N; ++i) {
    f(i);
  }
}

// this lambda implicitly converts to function pointer
for_upto(5, +[](int i) { std::cout << i << " "; });

// this lambda will not convert to function pointer
for_upto(5, [&values](int i) { values.push_back(i); });
```

std::function<>: Introduction (1/2)

- Standard library's class template std::function<> permits alternative formulation of for_upto

Template parameter of std::function<> specifies return type of function and its arguments
Can store anything having signature specified in std::function<> template parameter

```
void for_upto(int N, std::function<void(int)> f) {
  for (int i{0}; i != N; ++i) { f(i); }
}
```

std::function<>: Introduction (2/2)


```
std::vector<std::function<float(float,float)>> tasks;
tasks.push_back(std::fmaxf); // plain function
tasks.push_back(std::multiplies<float>()); // function object
tasks.push_back(std::multiplies<>()); // generic call operator

float x = 1.1f;
tasks.push_back([x](float a, float b) { return a*x+b; }); // lambda
tasks.push_back(
     [x](auto a, auto b) { return a*x+b; }); // generic lambda

// call each task
for (std::function<float(float, float)> f : tasks) {
  std::cout << f(10.1, 11.1) << "\n";
}
```

std::function<> and Member Functions (1/3)

- Core C++ language stops you from calling member function as non-member function: `std::string::length(str)`
- Can do so if member function stored in `function<>` object

```
std::string str{"C++: terror or horror"};
std::cout << std::string::length(str); // error
std::function<std::string::size_type(std::string const&)> f
                                       = &std::string::length;
std::cout << f(str);
```

std::function<> and Member Functions (2/3)


```
class C {
public:
  int func(int x, int y) const { return x*y; }
};

std::function<int(C const&,int,int)> mf = &C::func;
std::cout << mf(C(), 10, 20) << "\n";
```

std::function<> and Member Functions (3/3)

- Before:

```
std::vector<std::string> vs{"today", "is", "a", "good", "day"};
std::vector<int> vi;
std::transform(std::begin(vs), std::end(vs), std::back_inserter(vi),
 std::function<std::string::size_type(std::string const&)>
                                   (&std::string::length));
```

- After

```
std::vector<std::string> vs{"today", "is", "a", "good", "day"};
std::vector<int> vi;
std::transform(std::begin(vs), std::end(vs), std::back_inserter(vi),
               std::mem_fn(&std::string::length));
```

Sidebar: Pointers to [Member] Functions


```
int f(char, float);

struct Bart {
  //...
  static int f(char, float);
};

class Fred {
public:
  // ...
  int f(char, float);
};
```

```
int (*)(char, float);
```

```
int (Fred::*)(char, float);
```

`std::function<>`: Properties

- Generalized form of C++ function pointer with some fundamental operations
  - Can be used to invoke function without caller knowing anything about function itself
  - Can be copied, moved, and assigned
  - Can be initialized or assigned from another function (with compatible signature)
  - Has null state that indicates when no function is bound to it

`std::function<>` :Performance Considerations

- Generalized form of C++ function pointer with some fundamental operations
  - Lambdas can be inlined but not functions that're wrapped in `std::function<>`
  - `std::function<>` may use heap-allocated memory to store captured variables
  - Additional run-time overhead involved with calling functions wrapped in `std::function<>`

Generic Lambdas: Introduction

- Since C++14, lambda with at least one `auto` parameter is called *generic lambda*

```
auto v {3}; // int
auto lambda = [v] (auto v0, auto v1) {
  return v + v0*v1;
};
```

Generic Lambdas: Motivation (1/2)


```
std::vector<std::vector<std::string>> coll = {
  {"today", "is", "a", "good", "day"},
  {"tomorrow", "will", "be", "a", "better", "day"},
  {"however","I","cant","say","much","about","the", "day", "after"}
};

auto it2 = std::for_each(std::begin(coll), std::end(coll),
              [](std::vector<std::string> const& c) {
                     std::cout << c.size() << "\n"; });

auto it = std::find_if(std::begin(coll), std::end(coll),
          [](std::vector<std::string> const& c) {
                     return c.size() > 5; });
if (it != std::end(coll)) {
  print(*it, "*it: ");
}
```

Generic Lambdas: Motivation (2/2)


```
std::vector<std::vector<std::string>> coll = {
  {"today", "is", "a", "good", "day"},
  {"tomorrow", "will", "be", "a", "better", "day"},
  {"however","I","cant","say","much","about","the", "day", "after"}
};

auto it2 = std::for_each(std::begin(coll), std::end(coll),
          [](auto const& c) { std::cout << c.size() << "\n"; });

auto it = std::find_if(std::begin(coll), std::end(coll),
          [](auto const& c) { return c.size() > 5; });
if (it != std::end(coll)) {
  print(*it, "*it: ");
}
```

Generic Lambdas: Under the Hood (1/2)

- Lambda with at least one `auto` parameter is called *generic lambda*
- Generic lambdas can deduce generic parameter types not captured values
- Function call operator becomes member function template of closure

```
auto v {3}; // int
auto lambda = [v] (auto v0, auto v1) {
  return v + v0*v1;
};
```

Generic Lambdas: Under the Hood (2/2)


```
auto v {3}; // int
auto lambda =
 [v] (auto v0, auto v1) {
  return v + v0*v1;
};
```

```
class Lambda {
public:
  Lambda(int x) : v{x} {}
  template <typename T0, typename T1>
  auto operator()(T0 v0, T1 v1) const
  {
    return v + v0*v1;
  }
private:
  int v{};
};

auto v = 3;
auto lambda = Lambda{v};
```

Generic Lambdas: Instantiation

- Just like templated version, compiler won't generate actual function until lambda is invoked

```
// lambda definition ...
auto v {3};
auto lambda =  [v] (auto v0, auto v1) { return v + v0*v1; };

// instantiations caused by calls below ...
auto lambda_int = [v](int v0, int v1) { return v + v0*v1; };
auto lambda_dbl = [v](double v0, double v1) { return v + v0*v1; };

// calls ...
auto res_int = lambda_int(1, 2);
auto res_dbl = lambda_dbl(1.0, 2.0);
```

Generic Lambdas: Capture-Less

- Just like non-generic version, can convert generic capture-less lambda to function pointer

```
void f(void (*fp)(int))    { /*...*/ }
void g(void (*fp)(double)) { /*...*/ }

auto lam = [](auto x) {
    // generic code for x
};

// use lam as generic callback function pointer
f(lam);
g(lam);
```

Generic Lambdas: Recursion (1/2)

- Since lambda expression doesn't have specific type, recursive lambda expression must be wrapped in `std::function<>`

```
std::function<int(int,int)> power =
  [&power](int base, int exp) {
    return exp==0 ? 1 : base*power(base, exp-1);
  };

std::cout << power(2,10); // 2^10 = 1024
```

Generic Lambdas: Recursion (2/2)

- OTH, generic recursive lambda can take generic parameters without requiring `std::function<>`

```
// works on any numeric 'base' type
auto power = [](auto self, auto base, int exp) -> decltype(base)
{
  return exp==0 ? 1 : base*self(self, base, exp-1);
};

std::cout << power(power, 2, 10);       // 2^10 = 1024
std::cout << power(power, 2.71828, 10); // e^10 = 22026.3
```

auto Type Deduction (10)

- C++14 permits auto in lambda’s parameter to be deduced using template type deduction

```
std::vector<int> v{1,2,3}, v2{11,22,33};
auto reset_vec =
  [&v](auto const& new_val) {v = new_val;};

// ok
reset_vec(v2);
// error: cannot deduce type for {1,2,3}
reset_vec({11, 22, 33});
```

Variadic Lambdas: Introduction


```
auto print = [] (auto... params) {
  (std::cout << ... << params) << '\n';
};

print(1, 2, 3, "hello", 10.5);

// output: 123hello10.5
```

```
auto print = [] (auto... params) {
  ((std::cout << params << ", "), ...);
  std::cout << '\n';
};

print(1, 2, 3, "hello", 10.5);
// output: 1, 2, 3, hello, 10.5,
```

```
auto print = [] (auto first, auto... params) {
  std::cout << first;
  ((std::cout << ", " << params), ...);
  std::cout << '\n';
};

print(1, 2, 3, "hello", 10.5f);
// output: 1, 2, 3, hello, 10.5
```

Using Lambdas As Alternative To `std::bind<>`

- `std::bind<>` comes with cost of making code harder to optimize
- Turning all `std::bind<>` calls to lambda is simple
  - Turn any argument bound to variable or reference to variable into captured variable
  - Turn all placeholders into lambda arguments
  - Specify arguments bound to specific value directly in lambda body

Function Objects

- Have always existed in C++
- Also called *functionals* or *functors*
- Objects of class that defines `operator()`

```
class X {
public:
  // define function call operator
  return-value operator() (arguments) const noexcept;
  ...
};
```

Necessary if no exceptions are thrown

```
X func;
...
// a function call
func(arg1, arg2);
```

Required for function objects that don't change state

Why Function Objects?

- Faster than functions and function pointers
- “Smart” functions since they can be stateful
- See *wfo.cpp*

Types of Function Objects

- Zero parameter is called generator
  - See *gen.cpp*
- One parameter is called unary function
  - See *unary.cpp*
- Predicates are stateless function objects that return Boolean value
  - See *predicate.cpp*
- Two parameters is called binary function
  - See *binary.cpp*

Algorithms and Function Objects: Pass By Value (1/3)

- By default, function objects are passed to algorithms by value rather than by reference
- Advantage: You can pass any expression [*lvalue* or *rvalue*] of type function object

Algorithms and Function Objects: Pass By Value (2/3)


```
class IncreasingNumberGenerator {
  int num{};
public:
  IncreasingNumberGenerator(int ival) : num{ival} {}
  [[nodiscard]] int operator()() noexcept { return num++; }
};

IncreasingNumberGenerator seq(3);
std::list<int> li;
// insert sequence beginning with 3
std::generate_n(std::back_inserter(li), 5, seq);
// insert sequence beginning with 3 again ...
std::generate_n(std::back_inserter(li), 5,
                IncreasingNumberGenerator(3));
```

Algorithms and Function Objects: Pass By Value (3/3)

- Disadvantage: You can't get back modifications to state of function objects
- Three ways to get result from function objects passed to algorithms:
  - Keep state externally and let function object refer to it
  - Pass function objects by reference
  - Use return value of `for_each` algorithm

Pass By Reference


```
// passing fuction objects by reference ...
IncreasingNumberGenerator seq(3);
std::list<int> li;

// insert sequence beginning with 3
std::generate_n<std::back_insert_iterator<std::list<int>>,
               int, IncreasingNumberGenerator&>
                        (std::back_inserter(li), 5, seq);

print(li, "li: ");
// insert sequence beginning with 8 ...
std::generate_n(std::back_inserter(li), 5, seq);
```

Return Value of `for_each`

- See *foreach.cpp*

Generic Function Objects (1)

- Given list of people, you want to count number of people older than certain age

```cpp
// see people.h
class Person {
public:
  // other member functions
  int age() const;
private:
  // data members
};
```

```cpp
// see older-than.cpp
class older_than {
public:
  older_than(int limit) : m_limit {}
  bool operator()(Person const& p) const {
    return p.age() > m_limit;
  }
private:
  int m_limit;
};

std::vector<Person> vp {create_person_db()};
std::count_if(std::begin(vp), std::end(vp),
              older_than{50});
```

Generic Function Objects (2)

- Given list of people or cars or pets, you want to count how many items older than certain age
- Bad: Need to explicitly specify type of object!!!

```cpp
// see older-than-generic.cpp
template <typename T>
class older_than {
public:
  older_than(int limit)
      : m_limit{limit} {}
  bool operator()(T const& t) const {
    return t.age() > m_limit;
  }
private:
  int m_limit;
};
```

```cpp
// you can use older_than for any type
// that has getter age()
std::count_if(std::begin(persons),
              std::end(persons),
              older_than<Person>{50});

std::count_if(std::begin(pets),
              std::end(pets),
              older_than<Pet>{3});

std::count_if(std::begin(cars),
              std::end(cars),
              older_than<Person>{5});
```

Predefined Function Objects

- In `<functional>`, standard library provides several predefined function objects that cover fundamental operations

Arithmetic Functions

Predicate	Purpose
`std::plus<T>(x, y)`	`x + y`
`std::minus<T>(x, y)`	`x - y`
`std::multiplies<T>(x, y)`	`x * y`
`std::divides<T>(x, y)`	`x / y`
`std::modulus<T>(x, y)`	`x % y`
`std::negates<T>(x)`	`-x`

Comparison Predicates

Predicate	Purpose
`std::equal_to<T>(x, y)`	`x == y`
`std::not_equal_to<T>(x, y)`	`x != y`
`std::greater<T>(x, y)`	`x > y`
`std::less<T>(x, y)`	`x < y`
`std::greater_equal<T>(x, y)`	`x >= y`
`std::less_equal<T>(x, y)`	`x <= y`

Logical and Bitwise Predicates

<table>
<tr><th>Predicate</th><th>Purpose</th></tr>
<tr><td>std::logical_and<T>(x, y)</td><td>x && y</td></tr>
<tr><td>std::logical_or<T>(x, y)</td><td>x || y</td></tr>
<tr><td>std::logical_not<T>(x, y)</td><td>x > y</td></tr>
</table>

<table>
<tr><th>Predicate</th><th>Purpose</th></tr>
<tr><td>std::bit_and<T>(x, y)</td><td>x & y</td></tr>
<tr><td>std::bit_or<T>(x, y)</td><td>x | y</td></tr>
<tr><td>std::bit_xor<T>(x, y)</td><td>x ^ y</td></tr>
</table>

Predefined Function Objects: Sorting Criterion

- Typical use as sorting criterions
- Definition `std::set<int> coll;` expanded to

```
// sort elements with < in ascending order
std::set<int, std::less<int>> coll;
```

- Easy now to sort elements in opposite order:

```
// sort elements with > in descending order
std::set<int, std::greater<int>> coll;
```

Predefined Function Objects: Algorithms

- Another place to apply are algorithms

```
std::deque<int> d{1,2,3,5,7,11,13,17,19};

// negate all elements of d
std::transform(std::begin(d), std::end(d), // source
               std::begin(d),              // dest
               std::negate<int>());        // operation

// insert squared elements of d to d2
std::deque<int> d2;
std::transform(std::begin(d), std::end(d), // source1
               std::begin(d),              // source2
               std::back_inserter(d2),     // dest
               std::multiplies<int>());    // operation
```

What is a Function Adapter?

- Function adapter is function object that enables
  - *Partial function evaluation*: Creation of new function object from existing one by fixing one or more of its arguments to specific value(s)
  - *Functional composition*: Composition of function objects into more sophisticated function objects
- `std::bind<>` is most important adapter in standard library

Things To Do With `std::bind<>`

- `std::bind<>` allows you to:
  - Perform partial function evaluation and functional decomposition, i.e., adapt and compose new function objects out of existing function objects
  - Call global functions
  - Call member functions for objects, pointers to objects, and smart pointers to objects

What is `std::bind<>`?

- ~~Adapter that creates new function object from existing one by fixing one or more of its arguments to specific value~~
- ~~Concept called Partial Function Evaluation~~
- ~~If callable object requires some parameters, you can bind them to specific or passed arguments~~
  - ~~Specific arguments you simply name~~
  - ~~For passed arguments, you can use predefined placeholders~~
  - ~~Given the callable object and set of arguments, `std::bind<>` produces function object that can be called with "remaining" arguments, if any, of callable object~~

std::bind<>: Binding All Function Args To Specific Values (1/3)

cube<double> function isn't yet invoked; you've created function object that can be used to call cube<double> with specified argument 2.0

```
template <typename T> T cube(T);

auto cube2_dbl = std::bind(cube<double>, 2.0);
double d = cube2_dbl();
std::cout << d << '\n';

auto cube3_lng = std::bind(cube<long>, 3);
std::cout << cube3_lng() << '\n';
```

Only when calling the bound function object is the cubic value of 2.0 evaluated and returned

`std::bind<>`: Binding All Function Args To Specific Values (2/3)

- Consider following code fragment

```
auto bound = std::bind(std::greater<int>(), 5, 21);
bool is_5_gt_21 = bound(); // returns false
```

`std::bind<>`: Binding All Function Args To Specific Values (3/3)

`std::bind` creates a new function object

Function object saves a copy of function whose arguments it binds

It also creates copies of values function arguments are bound to

```
greater(int x, int y)
```

```
bound = std::bind(greater, 5, 21)
```

Function	1st arg	2nd arg
greater	5	21

```
bound()
```

```
greater(5, 51)
```

When this function object is called, it will call the stored function and pass it the saved arguments

std::bind<>: Binding Values to Specific Function Args (1/3)


```
void f(int x, std::string const& y) { // 2-ary function
  std::cout << "<" << x << " | " << y << ">\n";
}
```

bind f's 1st argument to 2 and its 2nd argument to g's (first) argument

```
using namespace std::placeholders;

auto g = std::bind(f, 2, _1);
g("hello");
// placeholder mechanism is quite flexible:
std::bind(f,_2,_1)("hello", 2); // also calls f(2, "hello")
std::bind(f,_1,_2)(2, "hello"); // also calls f(2, "hello")
std::bind(f,_1,"hello")(2);     // also calls f(2, "hello")
std::bind(f,2,_1)("hello");     // also calls f2(2, "hello")
```

calls f(2, "hello");

binds f's 1st and 2nd arguments to created function object's 2nd and 1st arguments, respectively and then calls bound function object …

std::bind<>: Binding Values to Specific Function Args (2/3)

bind greater's 1st argument to created function object is_gtr_42's (first) argument and 2nd argument to 42 …

```
using namespace std::placeholders;

auto is_gtr_42 = std::bind(std::greater<int>(), _1, 42);
auto is_les_42 = std::bind(std::greater<int>(), 42, _1);

is_gtr_42(6); // returns false
is_les_42(6); // returns true
```

bind greater's 1st argument to 42 and 2nd argument to created function object is_les_42's (first) argument …

std::bind<>: Binding Values to Specific Function Args (3/3)


```
greater(int x, int y)
```

```
is_gtr_42 = std::bind(greater, _1, 42)
```

Function	1st arg	2nd arg
greater	_1	42

```
is_gtr_42(6)
```

```
greater(6, 42)
```

```
is_les_42 = std::bind(greater, 42, _1)
```

Function	1st arg	2nd arg
greater	42	_1

```
is_les_42(6)
```

```
greater(42, 6)
```

Two-argument function tests whether its 1st argument is greater than its 2nd. Binding one argument to a value and other to placeholder creates unary function object that, when called with single argument, uses that argument to fill hole defined by placeholder.

std::bind<>: Reversing Arguments of Binary Function (1/2)

```
greater(int x, int y)
```

```
less_than = std::bind(greater, _2, _1)
```

Function	1st arg	2nd arg
greater	_2	_1

```
less_than(42, 6)
```

```
greater(6, 42)
```

_1 placeholder is filled with 1st argument passed to less_than, so the 1st argument to less_than becomes the 2nd argument to greater, and vice-versa

You specify 1st placeholder to be 2nd argument passed to greater, and vice-versa. This results in a new function less_than that's the same as greater, but with the arguments switched

std::bind<>: Reversing Arguments of Binary Function (2/2)

- How to sort a container in ascending order using `std::greater<>`?

```
std::vector<int> s{-10,10,-20,20,-30,30}, s2{s};

// sort in ascending order ...
std::sort(std::begin(s), std::end(s));

// sort in descending order ...
std::sort(std::begin(s), std::end(s), std::greater<int>());

// sort in ascending order ...
using namespace std::placeholders;
std::sort(std::begin(s2), std::end(s2),                // range
          std::bind(std::greater<int>(), _2, _1)); // predicate
```

Using `std::bind<>` On Functions With More Arguments (1)

- You want to use function with multiple arguments with standard algorithm that expects fewer arguments
- From *person.h* and *person-driver.cpp* …

Using `std::bind<>` On Functions With More Arguments (2)


```
// write Person data in multiple formats to output stream
void print_person(Person const& person,
                  std::ostream &os,
                  Person::OutputFormat fmt);

// create a database of Person ...
std::vector<Person> vp{create_person_db()};

// illegal because print_person needs 3 arguments while
// for_each only supplies a reference to Person
std::for_each(std::begin(vp), std::end(vp), print_person);
```

If `f` is function pointer copied by `for_each` and `it` is iterator pointing to a `Person` object in container `vp`, `for_each` does this: `f(*it)`
However, we want `for_each` to do this: `f(*it, os, fmt)`

Using std::bind<> On Functions With More Arguments (3)


```
// write Person data in multiple formats to output stream
void print_person(Person const& person,
                  std::ostream &os,
                  Person::OutputFormat fmt);

// create a database of Person ...
std::vector<Person> vp{create_person_db()};

using namespace std::placeholders;

// std::bind creates a unary function that specifies the
// output stream and format while leaving Person empty to
// be defined by std::for_each algorithm ...
std::for_each(std::begin(vp), std::end(vp),
              std::bind(print_person, _1,
                   ???  std::ref(std::cout), Person::BIO));
```

Using `std::bind<>` On Functions With More Arguments (4)


```
void print_person(Person const& person,
                  std::ostream &os,
                  Person::OutputFormat fmt);

std::for_each(std::begin(vp), std::end(vp),
              std::bind(print_person, _1,
                        std::ref(std::cout), Person::BIO));
```

1. `std::bind` stores *copies* of bound values in function object it returns.
2. Because copying is deleted on `std::cout`, you need to bind `os` argument to reference to `std::cout` and not its copy.
3. For this, you use `std::ref` helper function declared in `<functional>`.

Sidebar: Reference Wrappers (1/2)

- Class `reference_wrapper` declared in `<functional>` is used primarily to "feed" references to function templates that take their parameter by value [no need to specialize function templates]
- For given template parameter type `T`
  - Class provides `ref()` for implicit conversion to `T&`
  - Class provides `cref()` for implicit conversion to `T const&`
- Also works for ordinary functions
- Also allows references to be used as first-class objects such as in arrays and STL containers [not covered here]

Sidebar: Reference Wrappers (2)


```cpp
void incr(int &val) {
  ++val;
}

int i{5};

// increments copy of i
std::bind(incr, i)();

// increments i
std::bind(incr, std::ref(i))();
```

```cpp
template <typename T>
void foo(T val) {
  ++val;
}

int i{5};
// pass-by-value
foo(i);
// pass-by-reference
foo(std::ref(i));
// error: pass-by-const-reference
foo(std::cref(i));
```