Standard Library II

Function Objects (1)

Regular functions are not objects in C++

- Cannot be passed as parameters
- Cannot have state
- ...

C++ additionally defines the *FunctionObject* named requirement. For a type `T` to be a *FunctionObject*

- `T` has to be an object
- `operator()(args)` has to be defined for `T` for a suitable argument list `args` which can be empty
- Often referred to as *functors*

Function Objects (2)

There are a number of valid function objects defined in C++

- Pointers to functions
- Lambda expressions
- Stateful function objects in form of classes

Functions and function references are not function objects

- Can still be used in the same way due to implicit function-to-pointer conversion

Function Pointers (1)

While functions are not objects they do have an address

- Location in memory where the actual assembly code resides
- Allows declaration of *function pointers*

Function pointers to non-member functions

- Declaration: `return-type (*identifier)(args)`
- Allows passing functions as parameters
  - E.g. passing a custom compare function to `std::sort` (see later)
  - E.g. passing a callback to a method
- Can be invoked in the same way as a function

Function Pointers (2)

Example

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
    return (*func)(arg1, arg2);
}
//----------------------------------------------------------
double callFunc(double (*func)(double), double argument) {
    return func(argument);  // Automatically dereferenced
}
//----------------------------------------------------------
int add(int arg1, int arg2) { return arg1 + arg2; }
double add4(double argument) { return argument + 4; }
//----------------------------------------------------------
int main() {
    auto i = callFunc(add, 2, 4); // i = 6
    auto j = callFunc(&add4, 4);  // j = 8, "&" can be omitted
}
```

Lambda Expressions (1)

Function pointers can be unwieldy

- Function pointers cannot easily capture environment
- Have to pass all variables that affect function by parameter
- Cannot have “local” functions within other functions

C++ defines *lambda expressions* as a more flexible alternative

- Lambda expressions construct a closure
- Closures store a function together with an environment
- Lambda expressions can *capture* variables from the scope where they are defined

Lambda Expressions (2)

Lambda expression syntax

- `[` *`captures`* `] (` *`params`* `) ->` *`ret`* `{` *`body`* `}`
- *`captures`* specifies the parts of the environment that should be stored
- *`params`* is a comma-separated list of function parameters
- *`ret`* specifies the return type and can be omitted, in which case the return type is deduced from return statements inside the body

The list of captures can be empty

- Results in stateless lambda expression
- Stateless lambda expressions are implicitly convertible to function pointers

Lambda expressions have unique unnamed class type

- Have to use `auto` when assigning lambda expressions to variables
- Declaration of a lambda variable (e.g. as member) is not possible

Lambda Expressions (3)

Example

```
int callFunc(int (*func)(int, int), int arg1, int arg2) {
    return func(arg1, arg2);
}
//--------------------------------------------------------
int main() {
    auto lambda = [](int arg1, int arg2) {
        return arg1 + arg2;
    };

    int i = callFunc(lambda, 2, 4);  // i = 6
    int j = lambda(5, 6);            // j = 11
}
```

Lambda Expressions (4)

All lambda expressions have *unique* types

```
// ERROR: Compilation will fail due to ambiguous return type
auto getFunction(bool first) {
    if (first) {
        return []() {
            return 42;
        };
    } else {
        return []() {
            return 42;
        };
    }
}
```

Lambda Captures (1)

Lambda captures specify what constitutes the state of a lambda expression

- Can refer to *automatic variables* in the surrounding scopes (up to the enclosing function)
- Can refer to the `this` pointer in the surrounding scope (if present)

Captures can either capture *by-copy* or *by-reference*

- Capture by-copy creates a copy of the captured variable in the lambda state
- Capture by-reference creates a reference to the captured variable in the lambda state
- Captures can be used in the lambda expression body like regular variables or references

Lambda Captures (2)

Lambda captures are provided as a comma-separated list of captures

- By-copy: *`identifier`* or *`identifier initializer`*
- By-reference: `&`*`identifier`* or `&`*`identifier initializer`*
- *`identifier`* must refer to automatic variables in the surrounding scopes
- *`identifier`* can be used as an identifier in the lambda body
- Each variable may be captured only once

First capture can optionally be a capture-default

- By-copy: `=`
- By-reference: `&`
- Allows any variable in the surrounding scopes to be used in the lambda body
- Specifies the capture type for all variables without explicit captures
- If present, only diverging capture types can be specified afterwards

Lambda Captures (3)

Capture types

```
int main() {
    int i = 0;
    int j = 42;

    auto lambda1 = [i](){};      // i by-copy
    auto lambda2 = [&i](){};     // i by-reference

    auto lambda2 = [&, i](){};  // j by-reference, i by-copy
    auto lambda3 = [=, &i](){}; // j by-copy, i by-reference

    auto lambda4 = [&, &i](){}; // ERROR: non-diverging capture types
    auto lambda5 = [=, i](){};  // ERROR: non-diverging capture types
}
```

Lambda Captures (4)

Capture by-copy vs. by-reference

```
int main() {
    int i = 42;

    auto lambda1 = [i]() { return i + 42; };
    auto lambda2 = [&i]() { return i + 42; };

    i = 0;

    int a = lambda1();  // a = 84
    int b = lambda2();  // b = 42
}
```

Lambda Captures (5)

We can also capture a `this` pointer

- By-copy: `*this` (actually copies the current object)
- By-reference: `this`

```
struct Foo {
    int i = 0;

    void bar() {
        auto lambda1 = [*this]() {return i + 42; };
        auto lambda2 = [this](){ return i + 42; };

        i = 42;

        int a = lambda1();  // a = 42
        int b = lambda2();  // b = 84
    }
};
```

Lambda Captures (6)

⚠ By-copy capture-default copies only the this pointer

```
struct Foo {
    int i = 0;

    void bar() {
        auto lambda1 = [&]() {return i + 42; };
        auto lambda2 = [=](){ return i + 42; };

        i = 42;

        int a = lambda1();  // a = 84
        int b = lambda2();  // b = 84
    }
};
```

Lambda Captures (7)

Beware of lifetimes when capturing

```
#include <memory>

int main() {
    auto ptr = std::make_unique<int>(4);

    auto f2 = [inner = ptr.get()]() {
        return *inner;
    };

    int a = f2();  // 4
    ptr.reset();
    int b = f2();  // undefined behavior
}
```

By-reference capture can also easily lead to dangling references

Stateful Function Objects (1)

Situation so far

- Functions are generally stateless
- State has to be kept in surrounding object, e.g. class instances
- Lambda expressions allow limited state-keeping

Function objects can be implemented in a regular class

- Allows the function object to keep arbitrary state
- Difference to lambda expressions: State can be changed during lifetime

Stateful Function Objects (2)

Example

```
struct Adder {
    int value;

    int operator()(int param) {
        return param + value;
    }
};
//--------------------------------------------------------
int main() {
    Adder myAdder;
    myAdder.value = 1;
    myAdder(1);         // 2
    myAdder(4);         // 5
    myAdder.value = 5;
    myAdder(1);         // 6
}
```

std::function (1)

`std::function` is a general purpose wrapper for all callable targets

- Defined in the `<functional>` header
- Able to store, copy and invoke the wrapped target
- Potentially incurs dynamic memory allocations
- Often adds unnecessary overhead
- Should be avoided where possible

```
#include <functional>
//---------------------------------------------------------
int add2(int p){ return p + 2; }
//---------------------------------------------------------
int main() {
    std::function<int(int)>adder = add2;
    int a = adder(5); // a = 7
}
```

std::function (2)

Potential `std::function` use case

```
#include <functional>
//--------------------------------------------------------------
std::function<int()> getFunction(bool first){
    int a = 14;

    if (first)
        return [=]() { return a; };
    else
        return [=]() { return 2 * a; };
}
//--------------------------------------------------------------
int main() {
    return getFunction(false)() + getFunction(true)(); // 42
}
```

Working with Function Objects

Code that intends to call function objects should usually rely on templates

```
int bad(int (*fn)()) { return fn(); }
//------------------------------------------------------------
template <typename Fn>
int good(Fn&& fn) { return fn(); }
//------------------------------------------------------------
struct Functor {
    int operator()() { return 42; }
};
//------------------------------------------------------------
int main() {
    Functor ftor;

    bad([]() { return 42; });   // OK
    bad(ftor);                  // ERROR

    good([]() { return 42; });  // OK
    good(ftor);                 // OK
}
```

The Algorithms Library

The algorithms library is part of the C++ standard library

- Defines operations on ranges of elements `[first, last)`
- Bundles functions for sorting, searching, manipulating, etc.
- Ranges can be specified using pointers or any appropriate iterator type
- Spread in 4 headers
  - `<algorithm>`
  - `<numeric>`
  - `<memory>`
  - `<cstdlib>`
- We will focus on `<algorithm>` as it bundles the most relevant parts

std::sort

Sorts all elements in a range `[first, last)` in ascending order

- `void sort(RandomIt first, RandomIt last);`
- Iterators must be `RandomAccessIterators`
- Elements have to be swappable (`std::swap` or user-defined swap)
- Elements have to be move-assignable and move-constructible
- Does not guarantee order of equal elements
- Needs $O(n * log(N))$ comparisons

```
#include <algorithm>
#include <vector>
//----------------------------------------------------------
int main() {
    std::vector<unsigned> v = {3, 4, 1, 2};
    std::sort(v.begin(), v.end()); // 1, 2, 3, 4
}
```

Custom Comparison Functions

Sorting algorithms can be modified through custom comparison functions

- Supplied as function objects (`Compare` named requirement)
- Have to establish a strict weak ordering on the elements
- Syntax: `bool cmp(const Type1 &a, const Type2 &b);`
- Return `true` if and only if `a < b` according to some strict weak ordering <

```
#include <algorithm>
#include <vector>
//--------------------------------------------------------
int main() {
    std::vector<unsigned> v = {3, 4, 1, 2};
    std::sort(v.begin(), v.end(), [](unsigned lhs, unsigned rhs) {
        return lhs > rhs;
    }); // 4, 3, 2, 1
}
```

Further Sorting Operations

Sometimes `std::sort` may not be the optimal choice

- Does not necessarily keep order of equal-ranked elements
- Sorts the entire range (unnecessary e.g. for top-k queries)

Keep the order of equal-ranked elements

- `std::stable_sort`

Partially sort a range

- `std::partial_sort`

Check if a range is sorted

- `std::is_sorted`
- `std::is_sorted_until`

Searching

The algorithms library offers a variety of searching operations

- Different set of operations for sorted and unsorted ranges
- Searching on sorted ranges is faster in general
- Sorting will pay off for repeated lookups

Arguments against sorting

- Externally prescribed order that may not be modified
- Frequent updates or insertions

General semantics

- Search operations return *iterators* pointing to the result
- Unsuccessful operations are usually indicated by returning the `last` iterator of a range `[first, last)`

Searching - Unsorted

Find the first element satisfying some criteria

- `std::find`
- `std::find_if`
- `std::find_if_not`

Search for a range of elements in another range of elements

- `std::search`

Count matching elements

- `std::count`
- `std::count_if`

Many more useful operations (see reference documentation)

std::find

Example

```
#include <algorithm>
#include <vector>
//------------------------------------------------------------
int main() {
    std::vector<int> v = {2, 6, 1, 7, 3, 7};

    auto res1 = std::find(vec.begin(), vec.end(), 7);
    int a = std::distance(vec.begin(), res1); // 3

    auto res2 = std::find(vec.begin(), vec.end(), 9);
    assert(res2 == vec.end());
}
```

std::find_if

Example

```
#include <algorithm>
#include <vector>
//-----------------------------------------------------------
int main() {
    std::vector<int> v = {2, 6, 1, 7, 3, 7};

    auto res1 = std::find_if(vec.begin(), vec.end(),
        [](int val) { return (val % 2) == 1; });

    int a = std::distance(vec.begin(), res1); // 2

    auto res2 = std::find_if_not(vec.begin(), vec.end(),
        [](int val) { return val <= 7; });

    assert(res2 == vec.end());
}
```

Searching - Sorted

On sorted ranges, binary search operations are offered

- Complexity $O(log(N))$ when range is given as `RandomAccessIterator`
- Can employ custom comparison function (see above)

⚠ When called with `ForwardIterators` complexity is linear in number of iterator increments

Search for one occurrence of a certain element

- `std::binary_search`

Search for range boundaries

- `std::lower_bound`
- `std::upper_bound`

Search for all occurrences of a certain element

- `std::equal_range`

std::binary_search

Lookup an element in a range [first, last)

- Only checks for containment, therefore return type is bool
- To locate the actual values use std::equal_range

```
#include <algorithm>
#include <vector>
//-----------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto res1 = std::binary_search(v.begin(), v.end(), 3);
    assert(res1 == true);

    auto res2 = std::binary_search(v.begin(), v.end(), 0);
    assert(res2 == false);
}
```

std::lower_bound

Returns iterator pointing to the first element >= the search value

```
#include <algorithm>
#include <vector>
//------------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto res1 = std::lower_bound(v.begin(), v.end(), 3);
    int a = std::distance(v.begin(), res1);  // 3

    auto res2 = std::lower_bound(v.begin(), v.end(), 0);
    int b = std::distance(v.begin(), res2);  // 0
}
```

std::upper_bound

Returns iterator pointing to the first element $>$ the search value

```
#include <algorithm>
#include <vector>
//-------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto res1 = std::upper_bound(v.begin(), v.end(), 3);
    int a = std::distance(v.begin(), res1);  // 6

    auto res2 = std::upper_bound(v.begin(), v.end(), 4);
    assert(res2 == v.end());
}
```

std::equal_range

Locates range of elements equal to search value

- Returns pair of iterators (begin and end of range)
- Identical to using `std::lower_bound` and `std::upper_bound`

```
#include <algorithm>
#include <vector>
//-----------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 2, 3, 3, 3, 4};

    auto [begin1, end1] = std::equal_range(v.begin(), v.end(), 3);
    int a = std::distance(v.begin(), begin1);  // 3
    int b = std::distance(v.begin(), end1);    // 6

    auto [begin2, end2] = std::equal_range(v.begin(), v.end(), 0);
    assert(begin2 == end2);
}
```

Permutations

The algorithms library offers operations to permute a given range

- Can iterate over permutations in lexicographical order
- Requires at least `BidirectionalIterators`
- Values have to be swappable
- Order is determined using `operator<` by default
- A custom comparison function can be supplied (see above)

Initialize a dense range of elements

- `std::iota`

Iterate over permutations in lexicographical order

- `std::next_permutation`
- `std::prev_permutation`

std::iota

Initialize a dense range of elements

- std::iota(ForwardIt first, ForwardIt last, T value)
- Requires at least ForwardIterators
- Fills the range [first, last) with increasing values starting at value
- Values are incremented using operator++()

```
#include <numeric>
#include <memory>
//--------------------------------------------------------
int main() {
    auto heapArray = std::make_unique<int[]>(5);
    std::iota(heapArray.get(), heapArray.get() + 5, 2);

    // heapArray is now {2, 3, 4, 5, 6}
}
```

std::next_permutation

Reorders elements in a range to the lexicographically next permutation

- `bool next_permutation(BidirIt first, BidirIt last)`
- Returns `false` if the current permutation was the lexicographically last permutation (the range is then sorted in ascending order)

```
#include <algorithm>
#include <vector>
//--------------------------------------------------------
int main() {
    std::vector<int> v = {1, 2, 3};

    bool b = std::next_permutation(v.begin(), v.end());
    // b == true, v == {1, 3, 2}
    b = std::next_permutation(v.begin(), v.end());
    // b == true, v == {2, 1, 3}
}
```

std::prev_permutation

Reorders elements in a range to the lexicographically previous permutation

- `bool prev_permutation(BidirIt first, BidirIt last)`
- Returns `false` if the current permutation was the lexicographically first permutation (the range is then sorted in descending order)

```
#include <algorithm>
#include <vector>
//-----------------------------------------------------------
int main() {
    std::vector<int> v = {1, 3, 2};

    bool b = std::prev_permutation(v.begin(), v.end());
    // b == true, v == {1, 2, 3}
    b = std::prev_permutation(v.begin(), v.end());
    // b == false, v == {3, 2, 1}
}
```

Additional Functionality

The algorithms library offers many more operations

- `std::min` & `std::max` over a range instead of two elements
- `std::merge` & `std::in_place_merge` for merging of sorted ranges
- Multiple set operations (intersection, union, difference, ...)
- Heap functionality
- Sampling of elements using `std::sample`
- Swapping elements using `std::swap`
- Range modifications
  - `std::copy` To copy elements to new location
  - `std::rotate` To rotate range
  - `std::shuffle` To randomly reorder elements
- For even more operations: See the reference documentation

The Random Library

The random library defines pseudo-random number generators and distributions

- Defined in `<random>` header
- Bundles several useful components
  - Abstraction for random devices
  - Random number generators
  - Wrappers to generate numerical distributions from RNGs

Should *always* be preferred over functionality from `<cstdlib>` header

- `rand` produces very low-quality random numbers
- E.g. in one example the lowest bit simply alternates between `0` and `1`
- Especially serious if `rand` is used with modulo operations

Random Number Generators (1)

The random library defines various pseudo-random number generators

- Uniform pseudo-random bit generators with distinct properties
- RNGs can be seeded and reseeded
- RNGs can be equality-compared
- RNGs are *not* thread-safe
- Usually, one should prefer the Mersenne Twister generators

The random library additionally defines a `default_random_engine` type alias

- Implementation is implementation-defined

⚠ Do not use if you want portability

Most RNGs are template specializations of an underlying random number *engine*

⚠ Always use the predefined RNGs unless you know **exactly** what you are doing

Random Number Generators (2)

Mersenne Twister engine

- Predefined for 32-bit (`std::mt19937`) and 64-bit (`std::mt19937_64`) output width
- Produces high-quality unsigned random numbers in $[0, 2^w - 1]$ where w is the number of bits
- Can and should be seeded in the constructor

```
#include <cstdint>
#include <random>
//-------------------------------------------------------
int main() {
    std::mt19937 engine(42);

    unsigned a = engine();  // a == 1608637542
    unsigned b = engine();  // b == 3421126067
}
```

std::random_device

Standard interface to every available source of external randomness

- /dev/random, atmospheric noise, ...
- Actual sources are implementation dependent
- Only “real” source of randomness

⚠ Can degrade to a pseudo-random number generator when no source of true randomness is available

```
#include <cstdint>
#include <random>
//----------------------------------------------------------
int main() {
    std::mt19937 engine(std::random_device()());

    unsigned a = engine();  // a == ???
    unsigned b = engine();  // b == ???
}
```

Distributions

Random number generators are rather limited

- Fixed output range
- Fixed output distribution (approximately uniform)

The random library provides *distributions* to transform the output of RNGs

- All distributions can be combined with all random engines
- Various well-known distributions are provided
  - Uniform
  - Normal
  - Bernoulli
  - Possion
  - ...
- Some distributions are available as discrete or continuous distributions

std::uniform_int_distribution

Generates discrete uniform random numbers in range $[a, b]$

- Integer type specified as template parameter
- Constructed as `uniform_int_distribution<T>(T a, T b)`
- If not specified a defaults to 0 and b to the maximum value of T
- Numbers generated by `operator()(Generator& g)` where g is any random number generator

```
#include <random>
//---------------------------------------------------------
int main() {
    std::mt19937 engine(42);
    std::uniform_int_distribution<int> dist(-2, 2);

    int d1 = dist(engine);  // d1 == -1
    int d2 = dist(engine);  // d2 == -2
}
```

std::uniform_real_distribution

Generates continuous uniform random numbers in range $[a, b]$

- Floating point type specified as template parameter
- Constructed as `uniform_real_distribution<T>(T a, T b)`
- If not specified a defaults to 0 and b to the maximum value of T
- Numbers generated by `operator()(Generator& g)` where g is any random number generator

```
#include <random>
//-----------------------------------------------------------
int main() {
    std::mt19937 engine(42);
    std::uniform_real_distribution<float> dist(-2, 2);

    float d1 = dist(engine);  // d1 == -0.50184
    float d2 = dist(engine);  // d2 == 1.18617
}
```

Seeding

Random generators should generate new random numbers each time

- The seed value of a generator is used to calculate all other random numbers
- Normally the seed should itself be a random number, e.g. by `random_device`
- Deterministic sequences are preferable e.g. for tests or experiments
- For tests or experiments seed can be fixed to an arbitrary integer

⚠ Entropy of a generator is entirely dependent on the entropy of the seed generator

Generating Random Dice Rolls

Example

```
#include <random>
//-----------------------------------------------------------
int main() {
    // Use random device to seed generator
    std::random_device rd;
    // Use pseudo-random generator to get random numbers
    std::mt19937 engine(rd());
    // Use distribution to generate dice rolls
    std::uniform_int_distribution<> dist(1, 6);

    int d1 = dist(engine); // gets random dice roll
    int d2 = dist(engine); // gets random dice roll
}
```

Problems With Modulo

Modulo should in general *not* be used to limit the range of RNGs

- Most random number generators generate values in $[0, 2^w - 1]$ for some w
- When using modulo with a number that is not a power of two modulo will favor smaller values

Consider e.g. random dice rolls

- Assume a perfect random generator gen with $w = 3$
- gen will produce all values in $\{0, \ldots, 7\}$ with equal probability 0.125

```
int randomDiceroll() {
    return gen() % 6 + 1;
}
```

- $P(randomDiceroll() = x) = 0.25$ for $x \in \{1, 2\}$
- $P(randomDiceroll() = x) = 0.125$ for $x \in \{3, 4, 5, 6\}$