Programming in Modern C++

Module M47: C++11 and beyond: General Features: Part 2

Partha Pratim Das

Department of Computer Science and Engineering
Indian Institute of Technology, Kharagpur

ppd@cse.iitkgp.ac.in

All url's in this module have been accessed in September, 2021 and found to be functional

Module Recap

- Introduced following **C++11** general features:
  - auto
  - decltype
  - suffix return type (+ **C++14**)

Module Objectives

- Introducing following **C++11** general features:
  - Initializer List
  - Uniform Initialization
  - Range for Statement

Module Outline

Initializer Lists

Sources:

- Initializer lists, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- List-initialization (since C++11) and Constructors and member initializer lists, cppreference.com

Initializer Lists

- Consider:

```
vector<double> v = { 1, 2, 3.456, 99.99 }; // list of doubles
list<pair<string,string>> languages = {    // list of pairs of strings
    {"Nygaard", "Simula"}, {"Richards", "BCPL"}, {"Ritchie", "C"}
};
map<vector<string>,vector<int>> years = {  //list of vector<string>s and vector<int>s
    { {"Maurice", "Vincent", "Wilkes"}, {1913, 1945, 1951, 1967, 2000} },
    { {"Martin", "Ritchards"}, {1982, 2003, 2007} },
    { {"David", "John", "Wheeler"}, {1927, 1947, 1951, 2004} }
};
```

- Initializer lists are not just for arrays. The mechanism for accepting a {}-list (*braced list*) is a *function* (often a *constructor*) accepting an argument of type `std::initializer_list<T>`:

```
void f(initializer_list<int>);
f({1,2});
f({23, 345, 4567, 56789});
f({});  // the empty list
f{1,2}; // error: function call () missing
years.insert({{"Bjarne", "Stroustrup"}, {1950, 1975, 1985}});
```

Initializer Lists

- The initializer list can be of *arbitrary length*, but must be *homogeneous* (*all elements must be of the template argument type*, T, *or convertible to* T). An *initializer-list constructor* may be implemented as:

```
template<class E> class vector { public:
    vector(std::initializer_list<E> s) {              // initializer-list constructor
        reserve(s.size());                            // get the right amount of space
        uninitialized_copy(s.begin(), s.end(), elem); // init. elements (in elem[0:s.size())
        sz = s.size();                                // set vector size
    }
};
```

- The distinction between *direct* and *copy initialization* is maintained for {}-initialization, but is *less relevant*. For `std::vector` with an explicit ctor from `int` and an `initializer_list` ctor:

```
vector<double> v1(7);              // okay: v1 has 7 elements
v1 = 9;                            // error: no conversion from int to vector
vector<double> v2 = 9;             // error: no conversion from int to vector
void f(const vector<double>&);
f(9);                              // error: no conversion from int to vector
vector<double> v1{7};              // okay: v1 has 1 element (with its value 7.0)
v1 = {9};                          // okay v1 now has 1 element (with its value 9.0)
vector<double> v2 = {9};           // okay: v2 has 1 element (with its value 9.0)
f({9});                            // okay: f is called with the list { 9 }
vector<vector<double>> vs = {
    vector<double>(10),            // okay: explicit construction (10 elements)
    vector<double>{10},            // okay explicit construction (1 element with the value 10.0)
    10                             // error: vector's constructor is explicit
};
```

Initializer Lists

- The function can access the `initializer_list` as an *immutable sequence*. For example:

```
void f(initializer_list<int> args) {
    for (auto p = args.begin(); p != args.end(); ++p) cout << *p << "\n";
}
```

- `std::initializer_list` stores initializer values in an array and offers these member functions:
  - `size  // # of elements in the array`
  - `begin // ptr to first array element`
  - `end   // ptr to one-beyond-last array element`
- A constructor that takes a single argument of type `std::initializer_list` is called an **initializer-list constructor**
- The STL containers, `string`, and `regex` have initializer-list constructors, assignment, etc. An initializer-list can be used as a range, for example, in a `range for` statement (TBD later).
- The initializer lists are part of the scheme for *uniform and general initialization*. They also prevent *narrowing*
- Usually initializing using {} is preferred instead of () unless:
  - The code is shared with a C++03 compiler or
  - There is a need to use () to call a non-`initializer_list` overloaded constructor (rare)

Initializer Lists: std::initializer_list

- std::initializer_list looks something like: [Initializer Lists in C++ – std::initializer_list]

```
typedef unsigned int size_t; //#include <bits/C++config.h>
namespace std {
    template<class _E> class initializer_list { // initializer list
    public:
        typedef _E value_type;
        typedef const _E& reference;
        typedef const _E& const_reference;
        typedef size_t size_type;
        typedef const _E* iterator;
        typedef const _E* const_iterator;
    private:
        iterator _M_array;
        size_type _M_len;
        // The compiler can call a private constructor
        // constexpr defines compile-time constant expressions - TBD later
        constexpr initializer_list(const_iterator __a, size_type __l): _M_array(__a), _M_len(__l) { }
    public:
        constexpr initializer_list() noexcept: _M_array(0), _M_len(0) { }
        constexpr size_type size() const noexcept { return _M_len; }              // Number of elements
        constexpr const_iterator begin() const noexcept { return _M_array; }     // First element
        constexpr const_iterator end() const noexcept { return begin()+size(); } // One past last element
    };
};
```

Initializer Lists: initializer-list constructor


```
#include <iostream>
#include <string>
#include <vector>
#include <initializer_list>

template <typename T> // T is the type of initializer_list elements
class MyClass { std::vector<T> elems; /* vector to keep initialized values*/ public:
    // Default constructor
    MyClass(): elems({-1}) { std::cout << "Default Ctor: "; ShowElements(); }
    // Parametrized constructor
    MyClass(int b): elems({b}) { std::cout << "Parametrized Ctor: "; ShowElements(); }
    // Constructor using std::initializer_list
    MyClass(std::initializer_list<T> init_list): elems({init_list}) { // Using parenthesis
        // We can directly iterate on init_list as we do over elems
        std::cout << "Initializer List Ctor: "; ShowElements();
    }
    // Mixed Constructor
    MyClass(int i, std::initializer_list<T> init_list): elems{init_list} { // Without using parenthesis
        std::cout << "Mixed Ctor: " << i << ", "; ShowElements();
    }
    void ShowElements() /* Display the elements of elems */ { std::cout << "{ ";
        for (auto it = elems.begin(); it != elems.end(); ++it) std::cout << *it << ' ';
        std::cout << "}\n";
    }
};
```

Initializer Lists: initializer-list constructor


```
// template <typename T> class MyClass { std::vector<T> elems; public:
//     MyClass();                                          // Default constructor
//     MyClass(int b);                                     // Parametrized constructor
//     MyClass(std::initializer_list<T> init_list);        // Constructor using std::initializer_list
//     MyClass(int i, std::initializer_list<T> init_list); // Mixed Constructor
//     void ShowElements();
// };
int main() {
    MyClass<int> my_obj;                          /* my_obj{} */       // Default Ctor: { -1 }
    MyClass<int> my_obj_i  = MyClass<int>(500); /* my_obj_i(500) */  // Parametrized Ctor: { 500 }
    MyClass<int> my_obj_il = MyClass<int>{500}; /* my_obj_il{500} */ // Initializer List Ctor: { 500 }

    // initializer_list objects: std::initializer_list<int>
    auto init_list = { 1, 2, 3, 4, 5 };
    // May use init_list for { init_list }
    MyClass<int> my_obj_il_int = { init_list }; // Initializer List Ctor: { 1 2 3 4 5 }

    // initializer_list object
    std::initializer_list<std::string> il = { "Hello", "from", "PPD" };
    // May use il for { il }
    MyClass<std::string> my_obj_il_string = { il }; // Initializer List Ctor: { Hello from PPD }

    MyClass<std::string> my_obj_m = { 5, { "Thank", "You" } }; // Mixed Ctor: 5, { Thank You }
}
```

Initializer Lists: Overload Resolution

- Constructor with `std::initializer_list` parameter prefers {}-delimited arguments

```
class Widget { public:
    Widget(std::initializer_list<double> values); // #1
    Widget(double value, double uncertainty);     // #2
    ...
};
double d1, d2;
...
Widget w1 { d1, d2 }; // calls #1
Widget w2(d1, d2);    // calls #2
```

- Choose carefully between {} and () when initializing objects!

```
template <class T, class Allocator = allocator<T> > // from the C++11 standard
class vector { public: ...
    vector(size_type n, const T& value, const Allocator& = Allocator());
    vector(initializer_list<T>, const Allocator& = Allocator());
    ...
};
std::vector<int> v1(10, 5); // v1.size() == 10, all values == 5
std::vector<int> v2{10, 5}; // v2.size() == 2, values == {10, 5}
```

Initializer Lists: Overload Resolution

- Given multiple `std::initialization_list` candidates, best match is determined by worst element conversion:

```
class Widget { public:
    Widget(std::initializer_list<int>);          // #1
    Widget(std::initializer_list<double>);       // #2
    Widget(std::initializer_list<std::string>); // #3
    ...
};
Widget w1 { 1, 2.0, 3 };      // int => double same rank as double => int, so ambiguous
Widget w2 { 1.0f, 2.0, 3.0 }; // float => double better than float => int, so calls #2
std::string s
Widget w3 { s, "Init", "Lists" }; // calls #3
```

- If best match involves a narrowing conversion, call is invalid:

```
class Widget { public:
    Widget(std::initializer_list<int>); // #1
    Widget(int, int, int);              // #2
};
Widget w1 { 1, 2.0, 3 }; // Matches #1: error! double => int narrows
Widget w2 ( 1, 2.0, 3 ); // Matches #2: okay. double => int narrowing allowed
```

Initializer Lists: Braced Initializers and auto

- auto deduces std::initializer_list for braced initializers:

```
auto i = { 2, 4, 6, 8 };      // i is std::initializer_list<int>
```

- In general, templates deduce no type for braced initializers:

```
template<typename T> void f(T param) { ... }
f({ 2, 4, 6, 8}); // error! no type deduced for { 2, 4, 6, 8 }
```

  - Only way that auto type deduction ≢ template type deduction

- Especially for single-element braced initializers, this can confuse:

```
auto i1 = 10;                 // i1 is int
auto i2(10);                  // i2 is int
auto i3 {10};                 // i3 is std::initializer_list<int>
```

- Particularly when such variables interact with overload resolution:

```
std::vector<int> v1(i1); // v1.size() == 10, values == 0
std::vector<int> v2(i2); // v1.size() == 10, values == 0
std::vector<int> v3(i3); // v1.size() == 1, value == 10
```

- Use care when initializing auto variables with braced initializers!

Uniform Initialization: Syntax and Semantics

Sources:

- Uniform initialization syntax and semantics, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Initialization and Constructors and member initializer lists, cppreference.com

Uniform Initialization Syntax

- C++03 offers multiple initialization forms
  - Initialization $\neq$ assignment. For example, `const` objects can be initialized, not assigned
- Examples:

```
const int y(5);                    // direct initialization syntax
const int x = 5;                   // copy initialization syntax
int arr[] = { 5, 10, 15 };         // brace initialization
struct Point1 { int x, y; };
const Point1 p1 = { 10, 20 };      // brace initialization
class Point2 { public: Point2(int x, int y); };
const Point2 p2(10, 20);           // function call syntax
```

- Containers require another container:

```
int vals[] = { 10, 20, 30 };
const std::vector<int> cv(vals, vals+3); // init from another container
```

- Member and heap arrays are impossible:

```
class Widget {
    public: Widget(): data(???) {}
    private: const int data[5];          // not initializable
};
const float * pData = new const float[4]; // not initializable
```

Uniform Initialization Syntax

- Brace initialization syntax now allowed *everywhere*:

```
const int val1 {5};
const int val2 {5};
int a[] { 1, 2, val1, val1+val2 };
struct Point1 { int x, y; };
const Point1 p1 {10, 20};
class Point2 { public: Point2(int x, int y); };
const Point2 p2 {10, 20};                          // calls Point2 ctor
const std::vector<int> cv { a[0], 20, val2 };
class Widget {
    public: Widget(): data {1, 2, a[3], 4, 5} {}
    private: const int data[5];
};
const float * pData = new const float[4] { 1.5, val1-val2, 3.5, 4.5 };
```

- Really, *everywhere*:

```
Point2 makePoint() { return { 0, 0 }; }     // return expression; calls Point2 ctor

void f(const std::vector<int>& v);          // function declaration
f({ val1, val2, 10, 20, 30 });              // function argument
```

Uniform Initialization Semantics

- Semantics differ for aggregates and non-aggregates:
  - Aggregates (for example, arrays and structs):
    - Initialize members/elements beginning-to-end
  - Non-aggregates:
    - Invoke a constructor

Uniform Initialization Semantics: {}-Initializing Aggregates & Non-Aggregates

- **Aggregates**: Initialize members/elements beginning-to-end
  - Too many initializers ⇒ *error*
  - Too few initializers ⇒ remaining objects are *value-initialized*
    - ▷ Built-in types initialized to 0
    - ▷ UDTs with constructors are default-constructed
    - ▷ UDTs without constructors: members are value-initialized

```
struct Point1 { int x, y; };
const Point1 p1 = { 10 };                    // same as { 10, 0 }
const Point1 p2 = { 1, 2, 3 };               // error! too many initializers
```

    - ▷ `std::array` is also an aggregate:

```
long f();
std::array<long, 3> arr = { 1, 2, f(), 4, 5 }; // error! too many initializers
```

- **Non-Aggregates**: Invoke a constructor

```
class Point2 { public: Point2(int x, int y); }; short a, b;
...
const Point2 p1 {a, b};                      // same as p1(a, b)
const Point2 p2 {10};                        // error! too few ctor args
const Point2 p3 {5, 10, 20};                 // error! too many ctor args

std::vector<int> v { 1, a, 2, b, 3 };        // Holds for containers - calls a vector ctor
std::unordered_set<float> s { 0, 1.5, 3 }; // calls an unordered_set ctor
```

Uniform Initialization Semantics

- Brace-initialized variables may use =:

```
const int val1 = {5};
const int val2 = {5};
int a[] = { 1, 2, val1, val1+val2 };
struct Point1 { ... };
const Point1 p1 = {10, 20};
class Point2 { ... };
const Point2 p2 = {10, 20};
const std::vector<int> cv = { a[0], 20, val2 };
```

- Other uses of brace initialization cannot:

```
class Widget { // Not allowed in member initialization
    public: Widget(): data = {1, 2, a[3], 4, 5} {}                    // error!
    private: const int data[5];
};
// Not allowed in dynamic allocation
const float *pData = new const float[4] = { 1.5, val1-val2, 3.5, 4.5 }; // error!

// Not allowed in return
Point2 makePoint() { return = { 0, 0 }; }                             // error!

// Not allowed in function parameters
void f(const std::vector<int>& v);
f( = { val1, val2, 10, 20, 30 });                                     // error!
```

Uniform Initialization Semantics

- And `T var = expr` syntax cannot call `explicit` constructors:

```
class Widget {
public:
    explicit Widget(int);
    ...
};
Widget w1(10);     // okay, direct init: explicit ctor callable
Widget w2{10};     // okay, direct init: explicit ctor callable
Widget w3 = 10;    // error! copy init: explicit ctor not callable
Widget w4 = {10};  // error! copy init: explicit ctor not callable
```

- Develop the habit of using brace initialization without `=`
- Uniform initialization syntax a feature *addition*, not a replacement
  - Almost all initialization code valid in C++03 remains valid
    - Rarely a need to modify existing code

Uniform Initialization Semantics: {}-Initialization and Implicit Narrowing

- Sole exception: implicit narrowing
  - C++03 *allows it via brace initialization*, C++11 *does not*

```
struct Point { int x, y; };
Point p1 = { 1, 2.5 };                          // fine in C++03
                                                // implicit double => int conversion
                                                // error in C++11
Point p2 = { 1, static_cast<int>(2.5) }; // fine in both C++03 and C++11
```

- Direct constructor calls and brace initialization thus differ subtly:

```
class Widget {
    public: Widget(unsigned u);
    ...
};
int i;
...
Widget w1(i);      // okay, implicit int => unsigned
Widget w2{i};      // error! int => unsigned narrows
unsigned u;
Widget w3(u);      // fine
Widget w4{u};      // also fine, same as w3's init.
```

Uniform Initialization Summary

- Brace initialization syntax now available everywhere
  - *Aggregates* initialized *top-to-bottom / front-to-back*
  - *Non-aggregates* initialized via *constructor*
- *Implicit narrowing* not allowed.
- `std::initializer_list` parameters allow *initialization* lists to be passed to functions
  - Not actually limited to initialization (for example, `std::vector::insert`)
- Choose carefully between {} and () when initializing objects
  - Remember that `auto + { expr }` yields `std::initializer_list`

Range-for Statement

Sources:

- Range-for statement, isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Range-based for loop in C++

Ways of traversing a `vector`: Recap (Module 44)

- Let us revisit ways for the traversal of a vector as a sample container:

```
vector<int> v;

// subscript style
for(int i = 0; i < v.size(); ++i) { /* use v[i] */ } // native int for size
for(vector<T>::size_type i = 0; i < v.size(); ++i) { /* use v[i] */ } // correct size_type

// iterator style
for(vector<T>::iterator p = v.begin(); p != v.end(); ++p) { /* use *p */ } // verbose
for(vector<T>::value_type x : v) { /* use x read-only */ } // range for [C++11]
for(auto& x : v) { /* use x read-write*/ }                 // range for [C++11]
```


- Comparing *subscript* and *iterator* styles:
  - The *subscript style* is used in essentially every language
  - The *subscript style* does not work for lists and non-linear data structures (in C++ and in most languages)
  - The *iterator style* is used in C (pointers only) and C++
  - The *iterator style* is used for standard library algorithms
  - While both styles work for vectors, *iterator style* is more generic – works for all sequences

Range-for Statement

- A range for statement allows us to iterate through a *range*, which is anything we can iterate through like an STL-sequence defined by a `begin()` and `end()`
- All standard containers can be used as a range, as can a `std::string`, an initializer list, an array, a `valarray`, and any UDT that supports `begin()` and `end()`, for example, an `istream`:

```
void f(vector<double>& v) {
    for (auto x : v) cout << x << endl; // auto is vector<double>::value_type
    for (auto& x : v) ++x;  // using a reference to allow us to change the value
}
```

- A range `for` is read as `for all x in v` going through starting with `v.begin()` and iterating to `v.end()`:

```
for (const auto x : { 1, 2, 3, 5, 8, 13, 21, 34 })
    cout << x << endl;
```

- volatile may also be used:

```
    for (volatile int i : v) someOtherFunc(i); // or volatile auto i
```

- The `begin()` (and `end()`) can be a member to be called as `x.begin()` or a free-standing function to be called as `begin(x)`. *The member version takes precedence*

Range-for Statement: Example


```
// Illustration of range-for loop using CPP code
#include <iostream>
#include <vector>
#include <map>
int main() {
    // Driver
    std::vector<int> v = { 0, 1, 2, 3, 4, 5 }; // Iterating over whole array
    for (auto i : v) std::cout << i << ' '; std::cout << std::endl;

    // the initializer may be a braced-init-list
    for (int n : { 0, 1, 2, 3, 4, 5 }) std::cout << n << ' '; std::cout << std::endl;

    int a[] = { 0, 1, 2, 3, 4, 5 }; // Iterating over array
    for (int n : a) std::cout << n << ' '; std::cout << std::endl;

    // Just running a loop for every array element
    for (int n : a) std::cout << "In loop" << ' '; std::cout << std::endl;

    std::string str = "PPD"; // Printing string characters
    for (char c : str) std::cout << c << ' '; std::cout << std::endl;

    std::map <int, int> MAP({{1, 1}, {2, 4}, {3, 9}}); // Printing keys and values of a map
    for (auto i : MAP) std::cout << ' ' << i.first << ", " << i.second << std::endl;
}
```

Module Summary

- Introduced following **C++11** general features:
  - Initializer List
  - Uniform Initialization
  - Range for Statement