Programming in Modern C++

Module M46: C++11 and beyond: General Features: Part 1

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

Weekly Recap

- Familiarized with I/O libraries in C and C++
- Learnt Generic Programming
- Familiarized with C++ Standard Library with specific focus to STL
- Familiarized with containers, iterators and algorithms

Module Objectives

- Getting familiar with **C++11** and beyond: **C++14**, **C++17**, **C++20**, ...
- Introducing following **C++11** general features:
  - auto
  - decltype
  - suffix return type (+ **C++14**)

Module Outline

Major C++11 / C++14 Features

Sources:

- C++11 Language Extensions — General Features, isocpp.org
- C++11, cppreference.com
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- How is C++ in 2019?, Quora, 2019
- C++20/17/14/11, github

C++ Standards

C++98	C++11	C++14	C++17	C++20
1998	2011	2014	2017	2020
Templates	Move Semantics	Reader-Writer Locks	Fold Expressions	Coroutines
STL with Containers and Algorithms	Unified Initialization	Generic Lambda Functions	constexpr if	Modules
Strings	auto and decltype		Structured Binding	Concepts
I/O Streams	Lambda Functions		std::string_view	Ranges Library
	constexpr		Parallel Algortihms of the STL	
	Multi-threading and Memory Model		File System Library	
	Regular Expressions		std::any, std::optional, and std::variant	
	Smart Pointers			
	Hash Tables			
	std::array			
ISO/IEC 14882:1998	ISO/IEC 14882:2011	ISO/IEC 14882:2014	ISO/IEC 14882:2017	ISO/IEC 14882:2020

Fixes on C++98: C++03: ISO/IEC 14882:2003, 2003

Latest Version as of Sep-21: C++20: ISO/IEC 14882:2020, 2020

Major C++11 Features: Core Language Features

- `auto` and `decltype`
- trailing (suffix) return type
- list initialization (`initializer_list`)
- uniform initialization: brace-or-equal initializers
- `enum class`: scoped enums
- `constexpr` and literal types
- `noexcept` specifier and operator
- `nullptr`
- `defaulted` and `deleted` functions
- delegating and inherited constructors
- range-`for` (based on Boost)
- `static_assert` (based on Boost)
- Unicode string literals
- user-defined literals
- **rvalue references** and **move semantics**
- **lambda expressions**
- **concurrency support**
- GC interface (removed in C++23)
- `long long`, `char16_t` and `char32_t`
- `final` and override
- type aliases
- variadic templates
- generalized (non-trivial) unions
- generalized PODs (trivial types and standard-layout types)
- attributes
- `alignof` and `alignas`

Major C++11 Features: Library features

Header	Library features
• <array>	• atomic operations library
• <atomic>	• emplace() and other use of rvalue references throughout all parts of the existing library
• <cfenv>	• std::unique_ptr
• <chrono>	• std::move_iterator
• <cinttypes>	• std::initializer_list
• <condition_variable>	• stateful and scoped allocators
• <cstdint>	• std::forward_list
• <cuchar>	• chrono & ratio library
• <forward_list>	• new algorithms: ◦ std::all_of, std::any_of, std::none_of, ◦ std::find_if_not, std::copy_if, std::copy_n, ◦ std::move, std::move_backward, ◦ std::random_shuffle, std::shuffle, ◦ std::is_partitioned, std::partition_copy, std::partition_point, ◦ std::is_sorted, std::is_sorted_until, std::is_heap, std::is_heap_until, ◦ std::minmax, std::minmax_element, ◦ std::is_permutation, ◦ std::iota, ◦ std::uninitialized_copy_n
• <future>	• Unicode conversion facets
• <initializer_list>	• thread library
• <mutex>	• std::function
• <random>	• std::exception_ptr
• <ratio>	• std::error_code and std::error_condition
• <regex>	• iterator improvements: std::begin, std::end, std::next, std::prev
• <scoped_allocator>	• Unicode conversion functions
• <system_error>	
• <thread>	
• <tuple>	
• <typeindex>	
• <type_traits>	
• <unordered_map>	
• <unordered_set>	

Major C++14 Features: Language + Library Post-Recording

Core Language

- Binary literals
- Generalized return type deduction
- `decltype(auto)`
- Generalized lambda captures
- Generic lambdas
- Variable templates
- Extended `constexpr`
- The [[`deprecated`]] attribute
- Digit separators

Library

- Shared locking
- User-defined literals for `std::` types
- `make_unique`
- Type transformation `_t` aliases

auto & decltype

Sources:

- auto and decltype isocpp.org
- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Placeholder type specifiers (since C++11) and decltype specifier, cppreference.com
- C++ auto and decltype Explained
- GETTING TYPE NAME AT COMPILE TIME, Adam Badura, code::dive conference 2018, Video, Presentation, Library

auto

- In C++03, auto designated an object with *automatic storage type*. That is now *deprecated*
  We must specify the type of an object at declaration though the declaration may include an initializer with type
- In C++11 auto variables get the type from their *initializing expression*:

```
auto x1 = 10;              // x1: int

std::map<int, std::string> m;
auto i1 = m.begin();       // i1: std::map<int, std::string>::iterator
```

- const/volatile and reference/pointer adornments may be added:

```
const auto *x2 = &x1;      // x2: const int*
const auto& i2 = m;        // i2: const std::map<int, std::string>&
```

- To get a const_iterator, use cbegin (or cend, crbegin, and crend) container function:

```
auto ci = m.cbegin();      // ci: std::map<int, std::string>::const_iterator
```

- Type deduction for auto is akin to that for template parameters:

```
template<typename T> void f(T t);
f(expr);          // deduce T's type from expr
auto v = expr; // do essentially the same thing for v's type
```

auto

- For variables *not explicitly* declared to be a *reference*:
  - Top-level consts / volatiles in the initializing type are *ignored*
  - *Array and function names* in initializing types decay to *pointers*

```
const std::list<int> li;
auto v1 = li;                  // v1: std::list<int>
auto& v2 = li;                 // v2: const std::list<int>&

float data[BufSize];
auto v3 = data;                // v3: float*
auto& v4 = data;               // v4: float (&)[BufSize]
```

- Both *direct and copy initialization syntax* are permitted

```
auto v1(expr);                 // direct initialization syntax
auto v2 = expr;                // copy initialization syntax
```

For auto, both syntaxes have the same meaning

- auto is closely related to decltype and has extensive use in templates and generic lambdas

```
template<class T, class U> void multiply(const vector<T>& vt, const vector<U>& vu) {
    // ...
    auto tmp = vt[i]*vu[i]; // Compiler knows the type of tmp: product of T by a U
    // ...
}
```

decltype

- decltype yields the *type of an expression without evaluating it*

```
int x, *ptr;
decltype(x) i1;            // i1's type is int
decltype(ptr) p1;          // p1's type is int*
std::size_t sz = sizeof(decltype(ptr[44])); // sz = sizeof(int); ptr[44] not evaluated
```

- Fairly intuitive, but some quirks, for example, parentheses can matter:

```
struct S { double d; };
const S* p;
decltype(p->d) x1;                  // double
decltype((p->d)) x2;                // const double&
```

  - Quirks rarely relevant (and can be looked up when necessary)

- Can simplify complex type expressions

```
void f(const vector<int>& a, vector<float>& b) {
    typedef decltype(a[0]*b[0]) Tmp; // Type deonted by int * float
    for (int i=0; i<b.size(); ++i) {
        Tmp* p = new Tmp(a[i]*b[i]);
        // ...
    }
    // ...
}
```

auto / decltype: Semantic Differences

- auto and decltype both infer types from expressions; but they semantically differ:

```
#include <iostream>

int main() {
    int a = 5;          // int
    int& b = a;         // int&
    const int c = 7;    // const int
    const int& d = c;   // const int&

    // auto never deduces adornments like cv-qualifer or reference
    auto a_auto = a; // int
    auto b_auto = b; // int
    auto c_auto = c; // int
    auto d_auto = d; // int

    // cv-qualifer or reference needs to be explicitly added
    auto& b_auto_ref = a;          // int&
    const auto c_auto_const = a; // const int

    // decltype deduces the complete type of the expression
    decltype(a) a_dt;     // int         // [C++14] decltype(auto) a_dt_auto = a; // int
    decltype(b) b_dt = b; // int&        // [C++14] decltype(auto) b_dt_auto = b; // int&
    decltype(c) c_dt = c; // const int   // [C++14] decltype(auto) c_dt_auto = c; // const int
    decltype(d) d_dt = d; // const int& // [C++14] decltype(auto) d_dt_auto = d; // const int&
}
```

auto / decltype: Determining compiler-deduced types in C++

- Compiler deduces types of expressions in various contexts:
  - In C++03, types are inferred for implicit conversions, templates, etc.
  - In C++11, in addition, types are inferred for `auto` and `decltype`
- **How can we know the type deduced by the compiler?**
  - In C++ type is inferred at compiler time[1] - no support to know the inferred type
  - Debug in an IDE and check the type. This is possible only if the program compiles
  - Use compiler errors: Errors shown for: [Programiz - C++ Online Compiler]
    - ▷ Incomplete template: [Determining types deduced by the compiler in C++]

```
template<typename T> class KnowType;

int arr[] = { 1, 2, 3 }; // int [3]
KnowType<decltype(arr)> arr_type; // error: aggregate 'KnowType<int [3]> arr_type'
                                  // has incomplete type and cannot be defined
```

    - ▷ Incomplete type: [Using 'auto' type deduction - how to find out what type the compiler deduced?]

```
int arr[] = { 1, 2, 3 }; // int [3]
decltype(arr)::_;                 // error: decltype evaluates to 'int [3]', which
                                  // is not a class or enumeration type
```

[1] C++ is statically typed (except for dynamic polymorphism where there is typeid support for type)

auto / decltype: Determining compiler-deduced types in C++

- We may also use `typeid` operator to know the type
  - Not a good idea as `typeid` is meant for dynamic type
  - The name of type retured by `typeid` is encoded

```
#include <bits/stdc++.h> // includes all standard library, but is not a standard header file of GNU C++
                         // DO NOT USE
using namespace std;

int main() {
    int x;                                          // int
    char y;                                         // char
    cout << typeid(x).name() << endl;               // i
    cout << typeid(y).name() << endl;               // c

    vector<int> vi;                                 // std::vector<int>
    vector<double> vd;                              // std::vector<double>
    cout << typeid(vi).name() << endl;              // St6vectorIiSaIiEE
    cout << typeid(vd).name() << endl;              // St6vectorIdSaIdEE

    auto it = vi.begin();                           // std::vector<int>::iterator
    decltype(vi.cbegin()) cit = vi.cbegin();        // std::vector<int>::const_iterator
    cout << typeid(it).name() << endl;              // N9__gnu_cxx17__normal_iteratorIPiSt6vectorIiSaIiEEEE
    cout << typeid(cit).name() << endl;             // N9__gnu_cxx17__normal_iteratorIPKiSt6vectorIiSaIiEEEE
}
```

Suffix / Trailing Return Type

Sources:

- An Overview of the New C++ (C++11/14), Scott Meyers Training Courses
- Function declaration, cppreference.com
- When to use decltype(auto) versus auto?, cplusplus.com

Suffix / Trailing Return Type

- We really need `decltype` if we need a type for something *that is not a variable*, such as a *return type*. Consider:

```
template<class T, class U>
??? mul(T x, U y) { return x*y; }
```

- How to write the *return type*? It is the *type of* `x*y` – but how can we say that? Use `decltype`?

```
template<class T, class U>
decltype(x*y) mul(T x, U y) { return x*y; } // scope problem! types of x and y not known
```

- That won't work because `x` and `y` are not in scope. So:

```
template<class T, class U>
(decltype*(T*)(0)**(U*)(0)) mul(T x, U y) { return x*y; } // ugly! and error prone
```

- Put the return type where it belongs, after the arguments:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

  We use the notation auto to mean *return type to be deduced or specified later*

Suffix / Trailing Return Type

- The *suffix syntax* is not primarily about *templates* and *type deduction*, it is really about *scope*

```
struct List {
    struct Link { /* ... */ };
    Link* erase(Link* p); // remove p and return the link before p
    // ...
};
List::Link* List::erase(Link* p) { /* ... */ }
```

- The first `List::` is necessary only because the scope of `List` is not entered until the second `List::`. Better:

```
auto List::erase(Link* p) -> Link* { /* ... */ } // No explicit qualification for Links
```

- To declare objects, `decltype` can replace `auto`, but more verbosely:

```
std::vector<std::string> vs;
auto i = vs.begin();
decltype(vs.begin()) i = vs.begin();
```

- Only `decltype` solves the template-return-type problem in C++11 (by Perfect Forwarding)
- `auto` is for everybody. `decltype` is primarily for template authors

Suffix / Trailing Return Type: C++14

- In C++11, we use suffix return type to specify return type of templates to be inferred:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

  This is unclean because the return expression has to be *repeated* within decltype

- In C++14, suffix type can be skipped and the return type is deduced directly:

```
template<class T, class U>
auto mul(T x, U y) { return x*y; }
```

  For compatibility, it still supports the suffix return type. Hence, the following is still valid:

```
template<class T, class U>
auto mul(T x, U y) -> decltype(x*y) { return x*y; }
```

- C++14, further introduces decltype(auto) for deducing the return type by the semantics of decltype and not the semantics of auto. *No suffix return type is allowed here*

```
template<class T, class U>
decltype(auto) mul(T x, U y) { return x*y; }
```

- We present an example to highlight the differences between auto and decltype(auto)

Suffix / Trailing Return Type: C++14: auto / decltype(auto)


```
#include <iostream>

// returns prvalue: plain auto never deduces to a reference. prvalue is a pure rvalue - TBD later
template<typename T> auto foo(T& t) { return t.value(); }

// return lvalue: auto& always deduces to a reference
template<typename T> auto& bar(T& t) { return t.value(); }

// return prvalue if t.value() is an rvalue
// return lvalue if t.value() is an lvalue
// decltype(auto) has decltype semantics (without having to repeat the expression)
template<typename T> decltype(auto) foobar(T& t) { return t.value(); }

int main() {
    struct A { int i = 0 ; int& value() { return i ; } } a;
    struct B { int i = 0 ; int value() { return i ; } } b;

    foo(a) = 20; // *** error: expression evaluates to prvalue of type int
    foo(b);      // fine: expression evaluates to prvalue of type int

    bar(a) = 20; // fine: expression evaluates to lvalue of type int&
    bar(b);      // *** error: auto& always deduces to a reference (int&) - bar(b) needs an initializer

    foobar(a) = 20; // fine: expression evaluates to lvalue of type int&
    foobar(b);      // fine: expression evaluates to prvalue of type int
}
```

Suffix / Trailing Return Type: auto / declspec(auto)

- **Recommendations**
  - auto
    - Use auto to return a *prvalue with type deduction*
    - Use auto& or const auto& to return an *lvalue with type deduction*
  - declspec(auto)
    - Use declspec(auto) to write *forwarding templates*
    - Using decltype(auto), the return type is as what would be obtained if the expression used in the *return statement were wrapped in* decltype
    - Without decltype(auto), the deduction follows rules of *template argument deduction*
- **Summary**

```
auto foo() {                   // rt = int
    int x = 0; return (x); // returning local variable by value. Fine
}

decltype(auto) bar() {      // rt = int&
    int x = 0; return (x); // returning local variable by reference. Trouble
}
```

Source: When to use decltype(auto) versus auto?

Module Summary

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