Assignment 1

Yinghao Wang

October 11, 2016

Abstract

I traversed through the changes in the C++ standards to gain a better understanding of the language. I acknowledge notable changes in C++11, C++14, and upcoming changes in C++17.

1 C++11

1.1 auto

Prior to C++11, the auto keyword was a storage class specifier like static, register, and extern. Local variables have automatic storage duration, so the usage of auto was redundant.

In C++11, it now functions as a sort of placeholder for type. It can help make code look cleaner, and it is very effective when the type is obvious or is not important. It can also decipher what type the value should be after arithmetic operations.

```
for (std::vector<std::string>::iterator it = container.begin(); it != container.end();
    ++it); // long, and hard to read
// now using auto:
for (auto it = container.begin(); it != container.end(); ++it); // short, understandable,
    and clean
auto a = 1; // int
auto b = 1.23; // float
auto c = 1L; // long
auto d = Dog() // Dog object
auto e = 1 + (b * 2) * b; // understands this should be a float

auto i = 0, *p = &i; // okay i is int, p is pointer to int
auto j = 0, q = 2.7; // error: j is int, q is double
```

1.2 decltype

In C++11, decltype was introduced. It can be used to complement auto by deducing the type of auto variables. It can also deduce the type for functions. decltype may allow one to forward a return type without knowing whether the type is a reference or value. It is also able to determine if it is top-level const. Its intuitive use can allow for many useful features to be implemented into a program.

```
int i = 5;
decltype(i); // int
struct A { double x; };
auto a = new A;
decltype(a->x); // double
int fcn1(); // protoype for func. that returns an int
decltype(fcn1()) l = fcn1();
printf("%s\n", typeid(decltype(i)).name()); // outputting the parsed decltype to string
    format

const int const_i = 0, &const_j = const_i;
```

```
decltype(const_i) i = 0; // i has type const int
decltype(const_j) j = i; // j has type const int& and is bound to i
decltype(const_j) k; // error: k is a reference and must be initialized
// decltype of a parenthesized variable is always a references
decltype((const_i)) l; // error: l is a reference and must be initialized
```

1.3 nullptr

A null pointer does not point to anything, and it is used to check if a pointer is null before using it. Uninitialized pointers often lead to run-time errors.

More common in older programs, one may use a preprocessor variable called NULL defined in cstdlib as 0. When initializing a pointer to NULL, it is equivalent to initializing it as 0.

Newly introduced in C++11, the literal nullptr has a special type that can be converted to any other pointer type. Generally it is better practice to use nullptr instead of NULL due to its consistency. NULL constants generally do not have a well defined value whereas nullptr does.

```
// common cases (not much difference)
int* p1 = nullptr; //

// usage in overloaded functions
void f(char const *p);
void f(int v);
f(NULL); // intending to call the char const *int constructor, but the int constructor is
    called
f(nullptr); // calls char const *int constructor as planned

// one may even have a nullptr value constructor
void f(nullptr_t);
```

1.4 lambdas

An exciting new feature in C++11 is the anonymous function, also known as the lambda or lambda expression. It is a type of callable, an object or expression that is able to apply the call operator. It may use variables local to the function if it specifies them in the brackets. One may also use implicit capturing where one just inputs an "=" in the brackets, and all the variables used in the lambda function are captured. It is a great tool to add as it makes code much cleaner and more maintainable.

```
// basic lambda without parameters
auto lambda_f = [] { return "hello"; };
printf("%s\n", lambda_f()); // outputs hello

// using parameters
auto smaller = [] (const int &a, const int &b) { return (a < b) ? a : b; };
printf("%d\n", smaller(1, 2)); // outputs 1

// using local variables (explicit captures)
int num1 = 1, num2 = 2;
auto add1to2 = [num1, num2] { return num1 + num2; }; // would have error if num1, num2
    not captured (listed in the brackets)
printf("%d\n", add1to2()); // outputs 3

// using implicit captures
int global_num5 = 5; // now able to capture variables outside of local scope
int main() {
  int num3 = 3, num4 = 4;
  auto add3to4to5 = [=] { return num3 + num4 + global_num5; }; // implicit capture with
      [=]
```

```
  printf("%d\n", add3to4to5()); // outputs 12
  return 0;
}
// seamless usage with for_each, the lambda taking the place of a function
std::vector<int> nums { 1, 2, 3, 4, 5 };
for_each (nums.begin(), nums.end(), [] (int n) {
   printf("%d\n", n); // prints each element in int vector nums
});
```

1.5 Range-Based for

One of the most convenient changes in C++11 must be the range-based statement. It iterates through elements of an expression. It is a great alternative to traversing via indices or using iterators which may create sloppiness.

```
// iterating through an array
int num[] = { 1, 2, 3, 4, 5 };
for (int n : nums) {
   printf("%d", n); // output 12345
}

// iterating through a string
std::string str("hello");
for (char c : str) {
   printf("%c", c); // output hello
}

// iterating through a vector container
std::vector<int> v_nums = { 1, 2, 3 ,4 ,5 };
for (int n : v_nums) {
   printf("%d", n); // output 12345
}
```

1.6 Smart Pointers

Dynamic memory management is difficult as it is one of the greatest problems in the C and C++ language. One may forget to free memory leading to memory leaks. One may also free memory when there are still pointers trying to access the content which may lead to disastrous results. C++11 introduces **smart pointers** types which make memory management safer and simpler. **Smart pointers automatically delete an object.** There are two kinds of smart pointers. A **shared_ptr** has multiple pointers pointing to the same object. The other smart pointer, **unique_ptr**, has a single pointer pointing to a single object. A subclass of the shared_ptr is the **weak_ptr**, which only points to an object if it still needed (a great solution for dangling pointers).

```
std::shared_ptr<std::string> p1; // points to a string object
p1 = std::make_shared<std::string>("hi"); // p1 now to the string object with the value
    "hi"
auto p2(p1); // p2 and p1 point to same object
std::cout << p1.use_count() << '\n'; // output 2, the amount of pointers to the object,
    when it is 0, the object calls its destructor and is destroyed
std::weak_ptr<std::string> p3(p2); // shared_ptr has weak reference
std::cout << p1.use_count() << '\n'; // output 2, weak_ptr does not influence the
    use_count
p1.reset(); // p1 points to nullptr, p2 points to the object, use_count is now 1
p2.reset(); // p1, p2 points to nullptr, since use_count is 0, the object is now
    destroyed (weak_ptr does not prevent destruction of object), p3 points to nullptr

std::unique_ptr<std::string> u_p1 = make_unique<std::string>("hello");
auto u_p2(u_p1); // error: cannot share object with unique_ptrs
```

```
auto u_p2 = std::move(u_p1); // good: u_p1 points to nullptr, u_p2 points to object with
    "hello"
```

2 C++14

2.1 Return type deduction

Improving on auto from C++11, C++14 allows functions to let the compiler determine the return type of a function. Mainly, like auto, this is to aid readability. However, it can also be used to write generics.

```
// readability example
auto foo() { // this line is much shorter now
    std::vector<std::list<std::string>> result();
    return result;
}

// generic programming example
template<typename Ts, typename Tt>
auto add(Ts s, Tt t) {
    return s + t; // interprets the return value of the generics added together
}
```

2.2 Digit Separators

A very small, but useful change in C++14, digit separators allow increased readability and understanding of one's code.

```
int a = 9999999;
int b = 9'999'999; // same values, however, much more obvious
```

2.3 Binary Literals

Also a small change, binary literals are another one of those small changes that seemed like it should be obvious when C++ was first created. As the name suggests, it allows for the declaration of binary literals. Now programmers no longer need to transform constant base 10 numerals for certain bitwise operations.

```
int a = 0b01010101;
int b = 0b10101011;
// bitset outputs the binary representation of the values
std::cout << std::bitset<8>(a | b) << '\n'; // output 11111111
std::cout << std::bitset<8>(a & b) << '\n'; // output 00000001
```

2.4 Generic lambdas

A change in which there is improvement to both auto and lambda. Lambdas now allow auto as types in both parameters and return types now. This could help in little cases of readability, but it allows for much greater generic programming as generic lambdas brings all the power to lambdas that auto brought to C++11.

```
auto lambda_f = [](auto a, auto b) { return a + b; };
int int1 = 1, int2 = 2;
double db1 = 1.1, db2 = 2.2;
lambda_f(int1, int2); // int: 3
lambda_f(db1, db2); // double: 3.3
lambda_f(int1, db2); // double: 3.2
```

3 C++17

3.1 auto templates

Non-type templates are being proposed in C++17. They will greatly enhance the readability, especially of the instantiation of templates.

```
// without auto templating
template <typename T, T t> struct S {};
S<decltype(v), v> s;

// with auto templating
template <auto t> stuct S;
S<x> s;
```

3.2 constexpr if

A bigger change proposed in C++17 would be the constexpr if statements. The motivation for this would be the replacements of multiple overloads by replacing them with well-formed condition branches. This way, one would not need to waste time with overloading, as well as having shorter more concise code.

References

Lippman, Stanley. *C++ Primer, Fifth Edition.* Addison-Wesley Professional, 2013. Print.

"C++14 Language Extensions". *ISOCPP*, https://isocpp.org/wiki/faq/cpp14-language

"constexpr if". *open-std*, 10 Feb. 2016, http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0128r1.html.

"Declaring non-type template arguments with auto". *open-std*, 4 Mar. 2016, http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0127r1.html.