CMPS 109 – Advanced Programming – Assignment 1

Max Zhao mlzhao@ucsc.edu 1466223

Base Case – C++98

Features Added in C++11

1. auto
    a. Before C++11, declaration of objects required the specification of its type, and the 'auto' keyword was used for storage duration specification. Now it can be used as a placeholder for a type, which may be useful the type of object is verbose or automatically generated. However, 'auto' cannot be used as the return type of a function.

2. nullptr
    a. C++11 introduced nullptr as a new keyword to serve as a null pointer constant. It replaces the bug-prone NULL macro and the literal 0 that have been used as null pointer substitutes for many years. Previously, the NULL macro and '0' was not even a pointer type – although they worked most of the time, they ran into strange and unexpected edge cases that would be difficult to debug. Nullptr is implicitly convertible and comparable to any pointer type, pointer-to-member type, or bool (as false), but cannot be implicitly converted to integral types.

3. lambda functions
    a. lambda functions, also called anonymous functions, let you define functions locally, at the place of the call. Lambda functions are often arguments being passed to higher-order functions, or used for constructing the result of a higher-order function that needs to return a function. Using the [] construct inside a function call's argument list indicates to the system that a lambda expression is begun. A lambda function's functionality is similar to as if you defined a function whose body is placed inside another function call. [capture clase] (parameters) -> return-type {body}

4. shared_ptr
    a. C++98 only defined one smart pointer class, the now deprecated auto_ptr. The new shared_ptr is used when ownership of a memory resource should be shared. Shared_ptr objects may own the same object, can share ownership of an object while storing a pointer to another object, and also own no objects. The object is destroyed and its memory is deallocated when either the last remaining shared_ptr owning the object is destroyed, or its last remaining shared_ptr is assigned another pointer.

5. unique_ptr
    a. Similar to shared_ptr – but should ony be used when ownership of a memory resource does not have to be shared. It doesn't have a copy constructor, but has a move constructor, so it can be transferred to another unique_ptr. This smart pointer retains sole ownership of an object through a pointer and destroys that object when the unique_ptr goes out of scope. No two unique_ptr instances can manage the same object. Simply, unique ptr should be the default pointer used by C++11 code, replacing "raw" pointers as much as possible.

6. Strongly-typed enums
    a. Prior to C++11, enums exported their enumerators in the surrounding scope, which sometimes lead to name collisions, in other words – you couldn't have two enums that shared the same name. They were implicitly converted to integral types and cannot have a user-specified underlying type. However, with the new 'enum class' keywords, they no longer export their enumerators in the surrounding scope, are no longer implicitly converted to integral types and can have a user-specified underlying type.

7. Uniform brace notation
    a. C++ has various initialization notations, ranging from parenthesized notation, equality notation, brace notation, and member initializers. Such a large variety of initialization leads to confusion, to both novices and the experienced. Prior to C++11, you were also unable to initialize POD array members and POD arrays allocated using new[]. C++11 now allows for intuitive container initialization as well as in-class initialization of data members.

8. Range-based for loop
    a. C++11 extends syntax of 'for' statements to allow for easy iteration over a range of elements through use of the ':' operator. Called the 'range-based for', this can be used to iterate over any C-style arrays, initializer lists, and type that has begin() and end() functions.

Features Added in C++14

1. Generic lambda functions
    a. C++11's introduction of lambda functions required that lambda parameters be declared with concrete types, but 'auto' type declarations are allowed in lambda parameters in C++14, namely allowing lambdas to work with any suitable type and perform the expected calculations. This is useful when creating lambdas that can be reused in different contexts.

2. [[deprecated]] attribute
    a. Developers often run into a problem of outdated code, or code rot. Instead of making the developer comment out or delete the code, C++14 provides a new attribute with a systematic way to address it. The attribute-token *deprecated* can be used to mark names and entities whose use is still allowed, but is discouraged for some reason. It is also possible to place a string literal message inside the attribute tag to describe why this name or entity might be deprecated.

3. Return type deduction
    a. C++14 allows all functions to deduce the return type based on the type of expression given to the return statement. C++11 only allowed this to lambda functions. In order to induce return type deduction, the function must be declared with 'auto' as the return type, and if multiple return expressions are used in the function's implementation, then they must all deduce the same type. Recursion can be used with this new feature, but the recursive call must occur after at least one return statement in the definition of the function.

4. Binary literals and digit separators
    a. C++14 introduced some nice syntactic improvements through the addition of binary literals and digit separators. Although they provide no change in functionality of code, they improve readability, in turn reducing bug counts and developer frustration. Binary literals start with the prefix '0b' and are followed by binary digits. Digit separators don't affect the evaluation of a number, they simply exist to make the developer's job easier and make the numbers easy to read. To create a digit separator, we use a single quote character.

5. Constexpr
    a. Constexpr functions are functions that can be executed at compile time run constant expressions, such as when instantiating a template with an integer argument. Prior to C++14, constexprs were limited to containing only one function in one return statement, but C++14 extends the functionality of constexpr functions to allowing conditional statements and loops, local variable declarations, allowing of constexpr variables to be templated, and also mutation of objects whose lifetime began with the constexpr evaluation.

Features to be Expected in C++17

1. std::variant
    a. std::variant introduces a much-needed type-safe union in C++. An instance of std::variant at any given time either holds a value of one of its alternative types, or it holds no value. Std::variant takes similar API when compared to boost:variant, which was introduced in 2002.

2. If statement updates
    a. In C++17, we are expected to be able to declare variables within an if statement. This has many uses – currently, the initializer is either declared before the statement and leaked into the ambient scope, or an explicit scope is used. With the new form, code can be written much more compactly, and improved scope control leads to a less bug-prone source code. The declared variable in an if statement is also valid in the else part of the if statement; this was possible before in C++14, but only for a single variable.

References

http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/
http://www.codeproject.com/Articles/570638/Ten-Cplusplus11-Features-Every-Cplusplus-Developer
https://en.wikipedia.org/wiki/C%2B%2B11
http://www.drdobbs.com/cpp/the-c14-standard-what-you-need-to-know/240169034
https://en.wikipedia.org/wiki/C%2B%2B14
https://meetingcpp.com/index.php/br/items/final-features-of-c17.html
https://en.wikipedia.org/wiki/C%2B%2B17