Christopher Huynh

CMPS109 - Karim Sobh

Assignment 1 - Due: January 18

Report on C++ Versions

1. C++11
    a. Lambda Expressions
        i. Lambda expressions allows us to create functions where they are used immediately. Lambda functions are usually used within another function, like a for_each( ) function. When Lambda expressions are used with an auto variable, the variable name can be used to run the Lambda function. Lambda expressions are beneficial because it is very convenient to create a Lambda expression instead of creating an entirely new function to be used.
        ii. Example: auto addNumbers = [] (int a, int b) { return a + b;}
    b. Automatic types
        i. In C++11, the old auto was changed to a new variable type. Automatic types allow you to declare a variable without specifying what type it is. Whenever the auto variable is initialized or assigned, the compiler automatically decides what the variable type is. This is beneficial for when a variable changes types many times or is generated from a template.
        ii. Example: auto count = 10; auto avg = 6.534; auto flag = true;

c. Nullptr

   i. The nullptr is the constant of the pointer literal type nullptr_t. Nullptr is used for similarly to NULL and 0. There is no difference between using nullptr, NULL, and 0. The benefit is for code readability, clarity, and also because sometimes NULL and 0 can return unexpected results.

   ii. Example: if(nullptr) { std::cout << “Is null”;}

d. Ranged based for loop

   i. Ranged based for loop allows for it to be easy to iterate over elements in a list. This works for arrays, initializer lists, and any type with begin() and end() functions. The benefit of this is that it makes it way more convenient to iterate over arrays. The previous would have either been using a normal for loop or a for_each loop with begin() and end().

   ii. Example: for(int& a: array) { a += 10;} //adds 10 to all values in array

e. Delegating Constructors

   i. Constructors are now able to delegate or call other constructors in C++11. In previous versions, constructors were unable call other constructors, which increased the amount of code used to do the same thing. The ability to delegate constructors creates inheritance and eliminates code repetition.

   ii. Example: Dog(int dogCount) : number(dogCount) {} //constructor

Dog( ) : Dog(0) {} // Delegating beginning constructor

f. Using sizeof( ) with members of objects

i. Previously sizeof( ) could only be used with types and objects. Now it can be used on the members within an object without an explicit object. This is beneficial for allocating memory for custom types within an object.

ii. Example: sizeof(Object::member) //returns sizeof the type of member

2. C++14

a. Binary Literals

i. Originally, C++ only supported decimal, octal, and hexadecimal literals. In order to use binary literals, you had to convert them to a supported base. Now, by beginning the binary number with 0b or 0B, C++ will now interpret the binary literal. This benefits the use of bitwise operators and binary in C++

ii. Example: int x = 0b1010; // equivalent to decimal 10

b. Digit separators

i. You can now use quotation marks to separate digits anywhere in code. The quotation marks do not create parsing ambiguities unlike commas and underscores. The benefit of this is that it increases readability of code and does not affect anything else.

ii. Example: int x = 1’000’000; //this is equivalent to 1000000

c. [[deprecated]] attribute

i. C++14 allows the use of the [[deprecated]] attribute. This allows entities to still be used but gives a warning on compilation that the entity is

deprecated. You can also provide a message to go with the deprecation warning to explain why it is deprecated. This benefits continuously integrated software where a newer version of a function is more optimized.

ii. Example; [[deprecated]] int addNumbers() //when addNumbers is used compiler provides notice of deprecated.

d. Tuple addressing via type

i. Even though tuples were introduced in C++11, you could only address elements within a tuple by its index. Now in C++14, you can address an element within a tuple by its type. This does not work with tuples with multiple elements of the same type. This is beneficial because it allows you to address elements in a tuple without knowing its index.

ii. Example: int x = get<int>(my_tuple); //returns int within my_tuple

3. C++17

a. Initialization within if and switch

i. You can now initialize variables inside the if and switch statements. This removes verbose code where variables must be initialized before the if/switch statement, for example in the case of checking if a variable was inserted into a map, you must insert and then capture the return value of the insertion. This is beneficial because it prevents variables from leaking into the surrounding scope.

b. Inline variables

    i. Inline variables are used similarly to inline functions. This simplifies static data members and allow the use of in-class definitions without requiring an out-of-class definition. This also allows the definition of variables within header files. This is beneficial as it allows us to easily create header-only libraries.

References

1. http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/
2. https://en.wikipedia.org/wiki/C%2B%2B11
3. https://en.wikipedia.org/wiki/C%2B%2B14
4. https://en.wikipedia.org/wiki/C%2B%2B17