Travis Takai

ttakai@ucsc.edu

1375886

C++11

1) Trailing return type
   a. Used in conjunction with auto to not throw error if data hasn’t been used or determined before a function is executed. In the case where a set of variables types have yet to be determined at run time the trailing return type can assist in declaring what the function will return without causing any errors. This can be especially helpful when accepting ambiguous data as parameters. Mostly it benefits the developer from encountering errors from user data being inputted.
2) std::array
   a. Used to encapsulate a fixed amount of one data type that is declared along with the size. A useful addition to the std library as it allows for containers of data that do not require memory management. With the array comes functions that can allow quick access to either the front or rear of the array as well as the capacity and filling it quickly.
3) auto specifier
   a. A way to declare a piece of data without knowing the initial type it will be at run time. The auto specifier is a huge improvement in allowing the developer to leave some operations ambiguous and allow the data to respond correctly. Using auto can enable code to accept different types of

data as parameters so there will be some amount of fault tolerance built in as well. This enables both the back and front end of components to function with minimal errors.

4) std::tuple
    a. A new addition to the std library that allows for the creation of tuples that can contain different data types and is of a fixed length. Tuples can host many types of information so they can be dynamically fed data without causing errors. The tuple can be accessed by individual elements in array style but not be limited to only data type. This allows for more complex processing of data in which the user can feed in many types of values.
5) std:forward_list
    a. A more efficient implementation of a *list* in which bidirectional operations are not required. Its implementation is uses a singly-linked list so it is only forward-traversable and does not support quick random access of elements. The efficiency of access is much better giving the developers a useful way to store data without losing performance. If data is ordered before it is inserted, then there is a guarantee that the correct element can be accessed.
6) Lambda expressions
    a. A type of function that can return a type that is still possibly unknown. The lambda function is also much more concise and provides for more space for non-boilerplate code. They can additionally be used in the place of an argument for a function so that the data returned can be computed in

place. This allows for more flexibility in terms of when and where the data that will go in to function will be processed. Used in conjunction with the auto specifier allows for development focused on the main objective rather than the focus on syntax

C++14

1. decltype(auto)
    a. A way to specify that you would like to preserve the reference capability without having to explicitly state whether or not the return type will be a reference or non-reference. This means that the auto specifier can be expanded in use to more general cases without having to anticipate whether the data is going to be a reference or not. Allows for functions to return either references or non-references.
2. Return Type Deduction
    a. The *auto* type can be applied to functions as a return type. Additional qualifiers such as * and & also be used to ensure that it is the specific type we want returned. Allows for functions to be more ambiguous and work for a number of parameters. This means that numbers can always retain the maximum amount of information without using more space than is needed by the variable.
3. Generic lambdas
    a. Allows for the use of auto in as a parameter of a lambda function instead of having to explicitly declare variable types. Generic lambdas are great when the type of variable to be used is a possible unknown

or is known to take on different types (eg float vs int). Used with the auto specifier for variables, a majority of a program can be written to encompass many different data types.

4. Variable templates
    a. A way to create a template of a given value that can be instantiated using the parameters defined. Can allow for better memory management when running with limited space. Can be used in conjunction with containers or lists to stamp out many different containers without the overhead of defining a new one every time.

C++17

1. std::variant
    a. An object that can be hold several different types. It is a type-safe version of a union. It can only hold one type at a given time. Similar idea to a variable template, it can be used to allow for better tuning a program to the objective without compromising previously written code.
2. if constexpr(expression)
    a. A conditional built to determine first if the data being checked is a constexpr and then to perform the action given. The variable must both pass the check as a constexpr and the expression in order to execute the following block of code. Can help determine whether a set of data will be mutable or not.

Bibliography

- Microsoft C++ Standard Library Overview, https://msdn.microsoft.com/en-us/library/ct1as7hw.aspx
- “The Biggest Changes in C++11 (and Why You Should Care)”, Danny Kalev, http://blog.smartbear.com/c-plus-plus/the-biggest-changes-in-c11-and-why-you-should-care/
- cppreference, http://en.cppreference.com/w/
- “The C++14 Standard: What You Need to Know”, Mark Nelson, http://www.drdobbs.com/cpp/the-c14-standard-what-you-need-to-know/240169034
- “C++14 Language Extensions”, https://isocpp.org/wiki/faq/cpp14-language
- “Final features of C++17”, Jens Weller, https://meetingcpp.com/index.php/br/items/final-features-of-c17.html
- http://stackoverflow.com/