Reference – LearnCPP.com – has many excellent C++ tutorials

The auto keyword

In C++11, the meaning of the auto keyword has changed, and it is now a useful declaration feature.

Consider:

```
double d = 5.0;
```

If C++ knows 5.0 is a double literal, why do we have to explicitly specify that d is actually a double? Sounds like Python doesn't it?

Starting with C++11, the auto keyword does just that.

When initializing a variable, the auto keyword can be used in place of the variable type to tell the compiler to infer the variable's type from the initializer's type.

This is called **automatic type deduction**.

For example:

```
auto d = 5.0; // 5.0 is a double literal, so it
is type double
auto i = 1 + 2; // 1 + 2 evaluates to an
integer, so it is int
```

This even works with the return values from functions:

```
int add(int x, int y)
{
    return x + y;
}

int main()
{
    auto sum = add(5, 6);
    return 0;
}
```

Note that this only works when initializing a variable upon creation.

Using auto in place of fundamental data types only saves a few keystrokes, but where the types get complex and lengthy, using auto can be very nice.

The auto keyword can’t be used with function parameters

Many programmers new to using auto try something like this:

```
void mySwap(auto &x, auto &y)
{
    auto z = x;
    x = y;
    y = z;
}
```

This won’t work, because the compiler can’t infer types for function parameters x and y at compile time.

Use function templates, not automatic type deduction, in this case. The exception to this is in C++14 for lambda expressions, which is an advanced C++ topic.

Automatic type deduction for functions in C++14

In C++14, the auto keyword was extended to be able to auto-deduce a function's return type. Consider:

```
auto add(int x, int y)
{
    return x + y;
}
```

Since x + y evaluates to an integer, the compiler will deduce this function should have a return type of int.

While this may seem neat, this syntax should be avoided for functions.

The return type of a function helps to document for the caller what a function is expected to return.

A good rule of thumb is that auto is okay to use when defining a variable, because the type of the variable is to the right side of the statement.

However, with functions, that is not the case -- there's no context to help indicate what type the function returns. A user would actually have to dig into the function body itself to determine what type the function returned. It's much less intuitive, and therefore more error prone.

Trailing return type syntax in C++11

C++11 also added the ability to use a **trailing return syntax**, where the return type is specified after the rest of the function prototype.

Consider the following function declaration:

```
int add(int x, int y);
```

In C++11, this could be equivalently written as:

```
auto add(int x, int y) -> int;
```

In this case, auto does not perform automatic type deduction -- it is just part of the syntax to use a trailing return type.

Why would you want to use this?

One nice thing is that it makes all of your function names line up:

```
auto add(int x, int y) -> int;
auto divide(double x, double y) -> double;
auto printSomething() -> void;
auto calculateThis(int x, double d) -> string;
```

But it is of more use when combined with some advanced C++ features, such as classes and the decltype keyword.

For now, we recommend the continued use of the traditional function return syntax.

Summary

Starting with C++11, the auto keyword can be used in place of a variable’s type when doing an initialization in order to perform automatic type deduction.

Other uses of the auto keyword should generally be avoided.

For Each Loops

C++11 introduces a new type of loop called a **for-each** loop (also called a **range-based for** loop) that provides a simpler and safer method for cases where we want to iterate through every element in an array (or other list-type structure).

For each loop examples

The *for each* statement has a syntax that looks like this:

```
for (element_declaration : array)
statement;
```

When this statement is encountered, the loop iterates through each element in array, assigning the value of the current array element to the variable declared in element_declaration.

For best results, element_declaration should have the same type as the array elements, otherwise type conversion occurs.

Here is a simple example that uses a *for-each* loop to print all of the elements in an array named fib:

```
#include <iostream>

int main()
{
  int fib[] = {1,1,2,3,5,8,13,21,34,55,89};
  for (int number : fib)
     cout << number << ' ';

  return 0;
}
```

This prints:

```
1 1 2 3 5 8 13 21 34 55 89
```

Let’s take a closer look at how this works. First, the *for loop* executes, and variable number is set to the value of the first element, which has value 1. The program executes the statement, which prints 1. And so on.

Note that variable number is not an array index. It’s assigned the value of the array element for the current loop iteration.

For each loops and the auto keyword

Because element_declaration should have the same type as the array elements, this is an ideal case in which to use the auto keyword, and let C++ deduce the type of the array elements.

Here's the above example, using auto:

```
#include <iostream>

int main()
{
 int fib[] = {1,1,2,3,5,8,13,21,34,55,89};
  for (auto number : fib)
    cout << number << ' ';

    return 0;
}
```

For each loops and references

In the for-each examples above, the element declarations are declared by value:

```
int array[5] = { 9, 7, 5, 3, 1 };
for (auto element: array)
    cout << element << ' ';
```

This means each array element is copied into variable element.

Copying array elements can be expensive. We can use references when a copy is not needed:

```
int array[5] = { 9, 7, 5, 3, 1 };
for (auto &element: array)
  cout << element << ' ';
```

In the above example, element will be a reference to the currently iterated array element, avoiding having to make a copy.

Also any changes to element affects the array, something not possible if element is a normal variable.

And, of course, it’s a good idea to make your element const if you’re intending to use it in a read-only fashion:

```
int array[5] = { 9, 7, 5, 3, 1 };
for (const auto &element: array)
   cout << element << ' ';
```

Rule: Use references or const references for your element declaration in for each loops for performance reasons.

Activity - Rewrite the following max scores example using auto and a for each loop

Rewrite this example using a *for each* loop:

```cpp
#include <iostream>
using namespace std;
int main()
{
  const int numStudents = 5;
  int scores[numStudents] = {84,92,76,81,56};
  int maxScore = 0; //largest score
  for (int student = 0; student < numStudents;
++student)
    if (scores[student] > maxScore)
      maxScore = scores[student];
  cout << "Max score was " << maxScore << endl;
  return 0;
}
```

For each loops and non-arrays

For each loops work with many kinds of list-like structures, such as vectors (e.g. std::vector), linked lists, trees, and maps.

```
#include <vector>
#include <iostream>
using namespace std;
int main()
{
 vector<int> fib = {1,1,2,3,5,8,13,21,34};
    for (const auto &number : fib)
        cout << number << ' ';

    return 0;
}
```

For each doesn’t work with pointers to an array

In order to iterate through the array, for-each needs to know how big the array is, which means knowing the array size. Because arrays that have decayed into a pointer do not know their size, for each loops do not work with them!

```
#include <iostream>

int sumArray(int array[])
{
    int sum = 0;
    for (const auto &number : array) // compile
 error, the size of array isn't known
        ……
```

Can I get the index of the current element?

For each loops do *not* provide a direct way to get the array index of the current element. This is because many of the structures that *for each* loops can be used with (such as linked lists) are not directly indexable!

Conclusion

For-each loops provide a superior syntax for iterating through an array when we need to access all of the array elements in forwards sequential order.

It should be preferred over the standard for loop in the cases where it can be used. To prevent making copies of each element, the element declaration should ideally be a reference.

Note that because *for each* was added in C++11, it won’t work with older compilers.

Quiz

This one should be easy.

1) Declare a fixed array with the following names: Alex, Betty, Caroline, Dave, Emily, Fred, Greg, and Holly. Ask the user to enter a name. Use a *for each* loop to see if the name the user entered is in the array.

Sample output:

```
Enter a name: Betty Betty was found.
Enter a name: Megatron Megatron was not
found.
```

Hint: Use string as your array type.

For_each loops

Parameters first, last

Input iterators to the initial and final positions in a sequence. The range used is [first,last), which contains all the elements between *first* and *last*, including the element pointed to by *first* but not the element pointed to by *last*.

Parameter fn

Unary function that accepts an element in the range as argument.This is usually a function pointer. Its return value, if any, is ignored.

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;

void myfunction (int i) {
  cout << ' ' << i; }

int main () {
  vector<int> myvector;
  myvector.push_back(10);
  myvector.push_back(20);
  myvector.push_back(30);
  cout << "myvector contains:";
  for_each(myvector.begin(),myvector.end(),
  myfunction);
return 0; }
```

Using this in our HashTable class.

```
void printString (string s) {
    cout << ' ' << s << "\t";
}

ostream & operator << (ostream &out, const
HashTable & H) {
    for ( auto i = 0; i < SIZE; i++) {
        auto temp = H.table[i];
        cout << "Index " << i << " : ";
        for_each (temp.begin(), temp.end(),
           printString);
        cout << endl;
   }
    return out;
}
```

Better yet, after checking syntax:

```
ostream & operator << (ostream &out, const
HashTable & H) {
    for (auto const & temp: H.table) {
        auto i = 0;
        cout << "Index " << i++ << " : ";
        for_each (temp.begin(),temp.end(),
           printString);
        cout << endl;
    }
    return out;

}
```