C++ Syntax Notes: The Modern Range-Based "for" Loop

Modern Range-Based "for" Loops

In recent versions of C++, there is a version of the **for** loop that automatically iterates over all of the things in a container. This is very useful when used with a standard library container, because you don't have to worry about trying to access memory outside of a safe range, for example: the loop will automatically begin and end in the right place.

for (*temporary variable declaration* : *container*) { *loop body* }

There's an important detail about the temporary variable. If you declare an ordinary temporary variable in the loop, it just gets a copy of the current loop item by value. Changes you make to that temporary copy won't affect the actual container!

```
1   #include <iostream>
2   #include <vector>
3   int main() {
4
5     // In the standard library, a std::vector is an array with
6     // Let's make a vector of ints and loop over the contents.
7     // The syntax for std::vector<> is discussed further in th
8
9     std::vector<int> int_list;
10    int_list.push_back(1);
11    int_list.push_back(2);
12    int_list.push_back(3);
13
14    // Automatically loop over each item, one at a time:
15    for (int x : int_list) {
16      // This version of the loop makes a temporary copy of ea
17      // list item by value. Since x is a temporary copy,
18      // any changes to x do not modify the actual container.
19      x = 99;
20    }
21
22    for (int x : int_list) {
23      std::cout << "This item has value: " << x << std::endl;
24    }
25
26    std::cout << "If that worked correctly, you never saw 99!"
27
28    return 0;
29  }
```

Run

Reset

Expected output:

```
This item has value: 1
This item has value: 2
This item has value: 3
If that worked correctly, you never saw 99!
```

If you make the temporary variable of a reference type, you can actually modify the current container item instead of just getting a copy. This modified example shows how:

```
#include <iostream>
#include <vector>
int main() {

  std::vector<int> int_list;
  int_list.push_back(1);
  int_list.push_back(2);
  int_list.push_back(3);

  for (int& x : int_list) {
    // This version of the loop will modify each item direct
    x = 99;
  }

  for (int x : int_list) {
    std::cout << "This item has value: " << x << std::endl;
  }

  std::cout << "Everything was replaced with 99!" << std::er

  return 0;
}
```

Run

Reset

Expected output:

```
This item has value: 99
This item has value: 99
This item has value: 99
Everything was replaced with 99!
```

There are more advanced ways to use this, too. For example, if you are iterating over large objects in a container, then even if you don't want to modify the objects, you might want to use a **reference to a constant** as the loop variable type to avoid making a temporary copy of a large object, which could otherwise be slow.

```cpp
#include <iostream>
#include <vector>
int main() {

  std::vector<int> int_list;
  int_list.push_back(1);
  int_list.push_back(2);
  int_list.push_back(3);

  for (const int& x : int_list) {
    // This version uses references, so it doesn't make any
    // However, they are read-only, because they are marked
    std::cout << "This item has value: " << x << std::endl;
    // This line would cause an error:
    //x = 99;
  }

  return 0;
}
```

Run

Reset

Expected output:

```
This item has value: 1
This item has value: 2
This item has value: 3
```

This will probably be very useful to you in the future if you continue to use advanced C++.