



UNIVERSITY
OF WOLLONGONG
IN DUBAI

A large, modern, multi-story building with a glass and concrete facade, illuminated at dusk. The building has a prominent palm tree in the foreground. The sky is a deep blue, and the building's interior lights are visible through the glass windows.

University of Wollongong in Dubai



UNIVERSITY
OF WOLLONGONG
IN DUBAI

CSCI251 Advanced Programming – Dr HC Lim/Dr Abdellatif Tchantchane

Coverage

- Review STL and Generic Programming
- Review of selected topics



Range-based for loop (C++)

- Programming language has a convenient way to write a for loop over a range of values.
- As of C++11, C++ has the same concept; you can provide a container to your “for loop”, and it will iterate over it.
- Conceptually: “...Executes statement repeatedly and sequentially for each element in expression....”
- Syntax:
 for (for-range-declaration : expression)
 statement
- The range-based for loop changed in C++17 to allow the begin and end expressions to be of different types. And in C++20, an init-statement is introduced for initializing the variables in the loop-scope.



Range-based for loop (C++)

```
C:\Users\hclim\Documents\CSCI251LectPrep\
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
1 2 3 4 5 6 7 8 9 10
end of integer array test

Press any key to continue . . .
```

```
moreSTL.cpp > main()
1 // range-based-for loop
2
3 #include <iostream>
4 #include <vector>
5
6 int main()
7 {
8     // Basic 10-element integer array.
9     int x[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
10
11     // Range-based for loop to iterate through the array.
12     for( int y : x ) { // Access by value using a copy declared as a specific type.
13         // Not preferred.
14         std::cout << y << " ";
15     }
16     std::cout << std::endl;
17
18     // The auto keyword causes type inference to be used. Preferred.
19
20     for( auto y : x ) { // Copy of 'x', almost always undesirable
21         std::cout << y << " ";
22     }
23     std::cout << std::endl;
24
25     for( auto &y : x ) { // Type inference by reference.
26         // Observes and/or modifies in-place. Preferred when modify is needed.
27         std::cout << y << " ";
28     }
29     std::cout << std::endl;
30
31     for( const auto &y : x ) { // Type inference by const reference.
32         // Observes in-place. Preferred when no modify is needed.
33         std::cout << y << " ";
34     }
35     std::cout << std::endl;
36     std::cout << "end of integer array test" << std::endl;
37     std::cout << std::endl;
38
39     return 0;
40 }
```

Range-based for loop (C++)

- More code:
using <vector> container

C:\Users\hclim\Documents\CSCI251LectPrep\wk08\moreSTL.exe

```
0.14159 1.14159 2.14159 3.14159 4.14159 5.14159 6.14159 7.14159 8.14159 9.14159
end of vector test
Press any key to continue . . .
```

```
moreSTL.cpp > main()
1 // range-based-for loop
2
3 #include <iostream>
4 #include <vector>
5
6 int main()
7 {
8
9     // Create a vector object that contains 10 elements.
10    std::vector<double> v;
11    for (int i = 0; i < 10; ++i) {
12        v.push_back(i + 0.14159);
13    }
14
15    // Range-based for loop to iterate through the vector.
16    for( const auto &j : v ) {
17        std::cout << j << " ";
18    }
19    std::cout << std::endl;
20    std::cout << "end of vector test" << std::endl;
21
22    return 0;
23 }
```



Range-based for loop (C++)

- More code:
using <string> container

```
C:\Users\hclim\Documents\CSCI251LectPrep\wk08\r
Here are the planets:
Mercury
Venus
Earth
Mars
Jupiter
Saturn
Uranus
Neptune
Pluto (a dwarf planet)
Press any key to continue . . .
```

```
1  // This program demonstrates the range-based for loop.
2  // using <string>
3
4  #include <iostream>
5  #include <string>
6
7  int main()
8  {
9      std::string planets[] = { "Mercury", "Venus", "Earth", "Mars",
10         "Jupiter", "Saturn", "Uranus",
11         "Neptune", "Pluto (a dwarf planet)" };
12
13     // Display the values in the array
14     std::cout << "Here are the planets: " << std::endl;
15
16     for (std::string val : planets)
17         std::cout << val << std::endl;
18
19     return 0;
20 }
21
22
```



Range-based for loop (C++)

- As the range-based for loop executes, its range variable contains only a copy of an array element. Hence, you cannot use a range-based for loop to modify the contents of an array unless you declare the range variable as a reference.
- Recall that a reference variable is an alias for another value. Any changes made to the reference variable are actually made to the value for which it is an alias.
- To declare the range variable as a reference variable, place an ampersand (&) in front of its name in the loop header.
- Next slide shows an example. It uses a range-based for loop to store user input data in an array



Range-based for loop (C++)

- Sample code to get user inputs and modify the container.

```
c:\Users\hclim\Documents\CSCI251LectPrep\wl
Enter an integer value: 10
Enter an integer value: 20
Enter an integer value: 30
Enter an integer value: 40
Enter an integer value: 50

Here are the values you entered:
10 20 30 40 50
Press any key to continue . . .
```

```
1  // This program uses a range-based for loop
2  // to modify the contents of an array.
3  #include <iostream>
4
5  int main()
6  {
7      const int SIZE = 5;
8      int numbers[SIZE];
9
10     // Get values for the array.
11     for (int &val : numbers)
12     {
13         std::cout << "Enter an integer value: ";
14         std::cin >> val;
15     }
16     // Display the values in the array.
17     std::cout << "\nHere are the values you entered:\n";
18
19     for (int val : numbers)
20         std::cout << val << " ";
21
22     std::cout << std::endl;
23     return 0;
24 }
25
```



Range-based for loop (C++)

- From previous slide, notice that in line 11 the range variable, `val`, has an ampersand (&) written in front of its name. This declares `val` as a reference variable (&).
- As the loop executes, the `val` variable will not merely contain a copy of an array element, but it will be an alias for the element itself. Therefore, any changes made to the `val` variable will actually be made to the array element it currently references.
- Notice, by contrast, that in line 19 there is no ampersand written in front of the range variable's name. This is because here there is no need to declare `val` to be a reference(&) variable here. This loop is simply displaying the array elements, not changing them.



Range-based for loop (C++)

- Note:
 - The range-based for loop can be used in any situation where you need to step through all the elements of an array or container, and you do not need to use the element subscripts (or index value/position).
 - It will not work, however, in situations where you need the element subscript for some purpose. It will also not work if the loop control variable is being used to access elements of two or more different arrays. In these situations, you need to use the regular for loop



Function pointers

- For more detailed look at function pointers, refer to course text, Chapter 18.
- There you will see more information on function pointers and how to use function template and function pointers as in codes shown here.



Function templates

- Sample code:

```
1  // Function template sample code
2  // Exercising the use of function pointers as callback functions
3  #include <iostream>
4  #include <vector>
5  #include <iostream>
6  #include <string>
7
8  template <typename T>
9  const T* find_optimum(const std::vector<T>& values, bool (*compare)(const T&, const T&)) {
10     if (values.empty()) return nullptr;
11     const T* optimum = &values[0];
12
13     for (size_t i = 1; i < values.size(); ++i) {
14         if (compare(values[i], *optimum)) {
15             optimum = &values[i];
16         }
17     }
18     return optimum;
19 }
20
```



Function templates

- Sample code, cont'd

```
21 // Comparison prototypes:
22 bool less(const int&, const int&);
23 template <typename T> bool greater(const T&, const T&);
24 bool longer(const std::string&, const std::string&);
25
26 int main()
27 {
28     std::vector<int> numbers{ 91, 18, 92, 22, 13, 43 };
29     std::cout << "Minimum element: " << *find_optimum(numbers, less) << std::endl;
30     std::cout << "Maximum element: " << *find_optimum(numbers, greater<int>) << std::endl;
31     std::vector<std::string> names{ "Moe", "Larry", "Shemp", "Curly", "Joe", "Curly Joe" };
32     std::cout << "Alphabetically last name: "
33     << *find_optimum(names, greater<std::string>) << std::endl;
34     std::cout << "Longest name: " << *find_optimum(names, longer) << std::endl;
35 }
36
```



Function templates

- Sample code, cont'd
- Output from codes;

```
Min element: 13
Maximum element: 92
Alphabetically last name: Shemp
Longest name: Curly Joe
Press any key to continue . . .
```

```
36
37 bool less(const int& one, const int& other) {
38     return one < other;
39 }
40
41 template <typename T>
42 bool greater(const T& one, const T& other) {
43     return one > other;
44 }
45
46 bool longer(const std::string& one, const std::string& other) {
47     return one.length() > other.length();
48 }
49
```



Function templates

- Detailed explanation from previous codes:
 - This function template generalizes the `find_maximum()` and `find_minimum()` functions. The function pointer you pass to the `compare` parameter determines which “optimum” the function returns. The type of `compare` forces you to pass a pointer to a function that takes two `T` values as input and returns a Boolean. This function is expected to compare the two `T` values it receives and evaluate whether the first one is somehow “better” than the second one. The higher-order `find_optimum()` then calls the given comparison function through its `compare` parameter and uses this to determine which out of all the `T` values in its vector argument is best, or optimal.



Function templates

- Detailed explanation from previous codes (cont'd)
 - The key point is that you, as the caller of `find_optimum()`, determine what it means for one T value to be better or more optimal than the other. If it's the minimum element you want, you pass a comparison function equivalent to the less-than operator, `<`;
 - if it's the maximum element you want, the compare callback should behave like the greater-than operator, `>`.



Function Object

- Much like pointers to data values, pointers to functions are low-level language constructs that C++ has inherited from the C programming language. And just like raw pointers, function pointers have their limitations, which can be overcome using an object-oriented approach.
- Note that smart pointers are the object-oriented answer to the inherently unsafe raw pointers. A similar technique exists where objects are used as a more powerful alternative to plain C-style function pointers.
- These objects are called function objects or functors (the two terms are synonymous). Like a function pointer, a function object acts precisely like a function; but unlike a raw function pointer, it is a full-fledged class type object—complete with its own member variables and possibly even various other member functions.



Function Object

- Basic Function Objects
 - A function object or functor is simply an object that can be called as if it were a function. The key in constructing one is to overload the function call operator, as was briefly introduced in object session.
 - To see how this is done, we will define a class of function objects that encapsulate this simple function:
 - `bool less(int one, int other) { return one < other; }`



Function Object

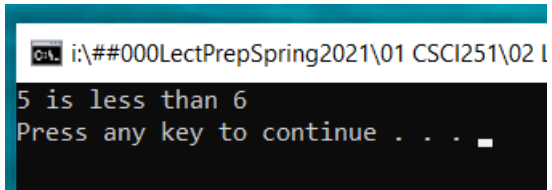
- This basic functor class has only one member: a function call operator. The main thing to remember here is that the function call operator is denoted with operator() and that its actual parameter list is specified only after an initial set of empty parentheses. Beyond that, this is just like any other operator function.
- You can define it in the corresponding source file in the usual manner, see codes on the right:

```
1  // Less.h - A basic class of functor objects
2  #ifndef LESS_H
3  #define LESS_H
4
5  class Less {
6  |   public:
7  |       bool operator()(int a, int b) const;
8  |   };
9
10 // Less.cpp - definition of a basic function call operator
11 bool Less::operator()(int a, int b) const {
12 |   return a < b;
13 |   }
14
15 #endif // LESS_H
16
```



Function Object

- Sample code:



```
i:\##000LectPrepSpring2021\01 CSCI251\02 I
5 is less than 6
Press any key to continue . . .
```

```
1 // Function object sample code
2 // Exercising the use of function objects
3 #include <iostream>
4 #include "Less.h"
5
6
7 int main()
8 {
9     Less less; // Create a 'less than' functor...
10    const bool is_less = less(5, 6); // ... and 'call' it
11
12    std::cout << (is_less? "5 is less than 6" : "Huh?") << std::endl;
13
14    return 0;
15 }
16
17
```



Function Object

- Of course, what is being “called” is not the object itself but rather its function call operator function. Note: You can also write `less(5,6)` as `less.operator()(5,6)`.
- Granted, creating a functor just to call it right after is not very useful at all. Things become a bit more interesting already if you use a functor as a callback function.
- To demonstrate this, you’ll first have to generalize a `find_optimum()` template (given in previous slide) for its callback argument ie use function object instead of function pointers.
- The most common way to generalize a function such as `find_optimum()` is therefore to declare a second template type parameter and to use that then as the type of the compare parameter:



Function Object

- Sample .h code

```
1 // Optimum.h - a function template to determine the optimum element in a given vector
2 #ifndef OPTIMUM_H
3 #define OPTIMUM_H
4 #include <vector>
5
6 template <typename T, typename Comparison>
7 const T* find_optimum(const std::vector<T>& values, Comparison compare)
8 {
9     if (values.empty()) return nullptr;
10    const T* optimum = &values[0];
11    for (size_t i = 1; i < values.size(); ++i)
12    {
13        if (compare(values[i], *optimum))
14        {
15            optimum = &values[i];
16        }
17    }
18    return optimum;
19 }
20
21 #endif // OPTIMUM_H
```



Function Object

- Sample program code:

```
Min i:\##000LectPrepSpring2021\01 CSCI251\02 Le
Minimum element: 13
Maximum element: 92
Press any key to continue . . .
```

```
1  // Exercising the use of a functor as callback functions
2  #include <iostream>
3  #include <vector>
4  #include "Optimum.h"
5  #include "Less.h"
6
7  template <typename T>
8  bool greater(const T& one, const T& other) { return one > other; }
9
10 int main()
11 {
12     Less less; // Create a 'less than' functor
13     std::vector<int> numbers{ 91, 18, 92, 22, 13, 43 };
14     std::cout << "Minimum element: " << *find_optimum(numbers, less) << std::endl;
15     std::cout << "Maximum element: " << *find_optimum(numbers, greater<int>) << std::endl;
16 }
17
```



note

- Because Comparison is a template type parameter, you can now invoke `find_optimum()` with compare arguments of any type you like. Naturally, the template's instantiation will then only compile if the compare argument is a function-like value that can be called with two T& arguments.
- And you know of two categories of arguments that might fit this bill already:
 - Function pointers of type `bool (*)(const T&, const T&)` (or similar types, such as for instance `bool (*)(T, T)`). Therefore, if you were to plug this new definition of `find_optimum<>()` this example would still work precisely like before.
 - Function objects of a type like `Less` that have a corresponding function call operator



Class templates

- Class templates are based on the same idea as function templates. A class template is a parameterized type; it's a recipe for creating a family of class types using one or more parameters. When you define a variable that has a type specified by a class template, the compiler uses the template to create a definition of a class using the template arguments that you use in the type specification. The argument for each parameter is typically (but not always) a type. You can use a class template to generate any number of different classes.
- It's important to keep in mind that a class template is not a class but just a recipe for creating classes because this is the reason for many of the constraints on how you define class templates.
- Refer to course text, chapter 16 for more details.



summary

- We have covered initial introduction to generic programming via templates
- We have reviewed previous topics on range-based for loop, function pointers and function objects and showed how they relate to generic programming.

