THE C++ STANDARD TEMPLATE LIBRARY

LECTURE 13-1

JIM FIX, REED COLLEGE CS2-S20

TODAY'S PLAN

- We'll survey the C++ *Standard Template Library* (or ***C++ STL***)
- We look at:
  - `vector`
  - `map` and `unordered_map`
  - "lambda" expressions

THE C++ STANDARD TEMPLATE LIBRARY (STL)

- A large collection of fully-realized C++ (templated) classes.
- Provides a lot of useful data types and functions.
  - container classes:
    - `vector`, `array`
    - `stack`, `queue`, `priority_queue`
    - `map`, `unordered_map`
  - `Iterator` for traversing these data structures
    - ➡ iterators are a generalized traversing pointer (or "handle")
  - `#include <algorithm>` for sorting, permuting, ...
    - supported by *lambdas*
  - "smart" pointers that provide better sharing and memory management

WHY USE THESE?

- Highly optimized.
- Mostly generic… carefully designed for lots of uses and for many contexts.
- Safe, debugged.
- Supported by the future evolution of the language.
- Adopted by modern C++ programmers.
  - Using them, your code will make sense to others.

A WORD ON MY PEDAGOGICAL CONFLICT

- This course had competing goals:
  - OT1H. Wanted to teach you low-level details...
    - ✦ machine representation: bits, bytes, assembly, GOTOs, ...
    - ✦ linking/arrays and memory organization
    - ✦ what's underneath standard data structures ("roll our own")
  - OTOH. Aspire to teach you how to engineer maintainable, readable, code
    - ✦ Mastering the STL might be the better approach to learning C++.
      - Can write more Pythonic code; but with compiled performance
- **So perhaps...**

  Mastering C++ requires you to learn the "raw" stuff, but you shouldn't use it.

maybe

AND SO...

- C++, for me, was a way of introducing you to low-level stuff.
- But others actually use it to engineer maintainable, readable, *correct* code.
  - ➡ Requires years of practice within C++ (and other languages, as well).
  - ➡ Using the C++ STL well is part of that practice.
- **NOTE:** you probably shouldn't be using C-style "raw" arrays.
  - The **`vector`** and **`array`** types in STL were intended to replace them.
- **NOTE:** you probably shouldn't regularly roll your own container data structures.
  - There are a wealth of STL ones for most common ones.
- **NOTE:** you probably should use pointers sparingly.
  - Should learn smart pointer classes **`shared_ptr`** and **`weak_ptr`**.

TODAY: LOOK AT VECTOR

- Here is a simple example of its use

```
#include <vector>
...

std::vector<int> iv {7,1,3,4,8};

// Output the elements using a "for" over the vector.
for (int x : iv) {
  std::cout << x << "\n";
}
std::cout << std::endl;

// Sum the elements using a "for" over the vector.
int sum = 0;
for (int x : iv) {
  sum += x;
}
std::cout << sum << std::endl;
```

FOR LOOP WITH AN ELEMENT REFERENCE

- Can also get a reference to each vector element

```
std::vector<int> iv {7,1,3,4,8};

// Update the elements using a "for" over the vector.
for (int& e : iv) { // Note the use of & here
  e = e + 10;
}

// This actually adds 10 to each vector component.
```

OPERATORS THAT LOOK LIKE ARRAY ACCESS

- Can use **`operator[]`** with **`vector`** like with a C-style array:

```
std::vector<int> iv {7,1,3,4,8};

// Output the elements by accessing each by an index.
for (int i = 0; i < iv.size(); i++) {
  std::cout << iv[i] << "\n";
}
std::cout << std::endl;

// Add 10 to each element
for (int i = 0; i < iv.size(); i++) {
  iv[i] = iv[i] + 10;
}
```

ITERATOR SYNTAX

- Can instead use iterators

```
std::vector<int> iv {7,1,3,4,8};

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
  std::cout << (*p) << "\n";
}
std::cout << "\n";

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
  (*p) += 10;
}
```

ITERATOR SYNTAX

- Can instead use iterators

```
std::vector<int> iv {7,1,3,4,8};

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
  std::cout << (*p) << "\n";
}
std::cout << "\n";

for (std::vector<int>::iterator p = iv.begin();
     p != iv.end();
     ++p) {
  (*p) += 10;
}
```

ITERATOR SYNTAX (DECLARED INSTEAD WITH AUTO)

- Have I told you yet about **`auto`**?????

```
std::vector<int> iv {7,1,3,4,8};

for (auto p = iv.begin(); p != iv.end(); ++p) {
  std::cout << (*p) << "\n";
}
std::cout << "\n";

for (auto p = iv.begin(); p != iv.end(); ++p) {
  (*p) += 10;
}
```

- The **`auto`** keyword lets C++ *infer* the type of the variable.
  - Many consider it good style; I personally use it only for tricky types.
  - (But **BTW/FYI**: I ***LOVE*** *type inference* in *other* languages.)

ITERATORS TIE NICELY WITH <ALGORITHM>

- Here is a sorting library function

```
#include <algorithm>

...

sort(iv.begin(), iv.end(), std::greater<int>());
```

- We're passing two iterators (pointer-ish things) to locations within a vector:
  - one for the beginning element, one for *just past* the ending element.
  - These give an *extent* (or *range*) within a container.
- The third argument is a function for comparing two **`int`** values
  - This is from a Stroustrup example; it sorts **`iv`** in reverse.

THE AT METHOD PERFORMS "BOUNDS CHECKING"

- **NOTE:** `operator[]` does not check the index.
- To have the class perform bounds checking use `at` instead.

```
std::vector<int> iv {7,1,3,4,8};
// Output the elements by accessing each by an index.
for (int i = 0; i < iv.size(); i++) {
  std::cout << iv.at(i) << "\n";
}
std::cout << std::endl;

// Add 10 to each element
for (int i = 0; i < iv.size(); i++) {
  iv.at(i) = iv.at(i) + 10;
}
```

GROWING A VECTOR

- C++ programmers often **`push_back`**

```
std::vector<int> iv {};
std::string entry;
do {
  std::cin >> entry;
  if (entry != "done") {
    int value = std::stoi(entry);
    iv.push_back(value); // puts at the end of the vector
} while (entry != "done");
```

- Under the covers C++ is maintaining a C-style array, resizing it occasionally.

GROWING A VECTOR; SHRINKING A VECTOR

- C++ programmers often **`push_back`**

```
std::vector<int> iv {};
std::string entry;
do {
  std::cin >> entry;
  if (entry != "done") {
    int value = std::stoi(entry);
    iv.push_back(value); // puts at the end of the vector
} while (entry != "done");
```

- Under the covers C++ is maintaining a C-style array, resizing it occasionally.
- There is also a method **`pop_back`**
  - ➡ This shrinks the **`vector`**, and the last element is removed.

RESIZING A VECTOR

- You can do several other things, e.g. you can **`resize`** it.

```
iv.resize(12);
```

- If the size is 5, then this essentially performs 7 "push backs."
  - ➡ It fills those 7 elements with the default value for its element type.

- **NOTE:** different than **`reserve`**, which adds capacity *under the covers*

```
iv.reserve(new_capacity);
```

EDITING WITHIN A VECTOR

- You can performs "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;
(*place) = 4567890;
```

➡ This modifies the item at index 6.

EDITING WITHIN A VECTOR

- You can performs "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;
(*place) = 4567890;
```

➡ This modifies the item at index 6.

EDITING WITHIN A VECTOR

- You can performs "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;
(*place) = 4567890;
```

  - ➡ This modifies the item at index 6.

- Can **`insert`** a new item, say, before the one at index 4:

```
iv.insert(iv.begin()+4,987);
```

- Can **`erase`** a chunk of items (like Python's `[2:-4]` range notation):

```
iv.erase(iv.begin()+2,iv.end()-4);
```

EDITING WITHIN A VECTOR

- You can performs "iterator arithmetic" to work within a vector:

```
std::vector<int>::iterator place = iv.begin()+6;
(*place) = 4567890;
```

  - ➡ This modifies the item at index 6.

- Can **`insert`** a new item, say, before the one at index 4:

```
iv.insert(iv.begin()+4,987);
```

- Can **`erase`** a chunk of items (like Python's `[2:-4]` range notation):

```
iv.erase(iv.begin()+2,iv.end()-4);
```

VECTOR STORAGE

- Here is a little test illustrating how storage is managed. Consider this class:

```
class Box {
public:
  int value;
  Box(int v) : value {v} { }
  void square() { value *= value; }
}
```

- Consider this client code:

```
Box a {4};
Box b {5};
Box c {6};
std::vector<Box> bv {a,b,c};
a.square(); // changes a, but not bv[0].
```

- Changing the contents of **a**, **b**, **c** does not change **bv** elements.
  - The vector **bv** has its own storage for each **Box** element.

VECTOR STORAGE OF BOX (CONT'D)

- Iterating can act on the contents of these boxes:

```
for (Box& e : bv) {
  std::cout << e.v << std::endl;
}
std::cout << std::endl;
```

```
for (Box& e : bv) {
  e.v += 200;
}

for (int i=0; i<bv.size(); i++) {
  std::cout << bv[i].v << std::endl;
}
std::cout << std::endl;

bv[1].square();

std::vector<Box>::iterator p = bv.begin()+2;
p->square();
```

OTHER CONTAINERS

- In addition to **`std::vector<T>`**, there is **`std::array<T>`**
  - Not dynamically resizeable, maintains fixed size.
- There is a **`std::list<T>`** for linked lists.
- There are two kinds of "associative" (i.e. *key/value storage*; a *dictionary*)
  - the **`std::map<K,V>`** container is, in essence, a binary search tree.
    - ➡ It's an *ordered dictionary*.
  - the **`std::unordered_map<K,V>`** container is a hash table.
    - ➡ It's an un*ordered dictionary*.

EXAMPLE USE OF STD::MAP

- Here is some code building a dictionary mapping strings to integers

```
#include <map>
...
std::map<std::string,int> m {};
m.insert(std::make_pair("Gwen", 49));
m.insert(std::make_pair("Carlos", 25));
m["Bob"] = 17;  // also inserts
m["Gwen"] = 50; // updates
std::map<std::string, int>::iterator p = m.find("Jamie");

while (auto q = m.begin(); q != m.end(); q++) {
  std::cout << q->first <<":"<< q->second << std::endl;
}
```

- The loop at the end outputs the entries in alphabetical order.

EXAMPLE USE OF STD::MAP

- Here is some code building a dictionary mapping strings to integers

```
#include <map>
...
std::map<std::string,int> m {};
m.insert(std::make_pair("Gwen", 49));
m.insert(std::make_pair("Carlos", 25));
m["Bob"] = 17;  // also inserts
m["Gwen"] = 50; // updates
std::map<std::string, int>::iterator p = m.find("Jamie");

while (auto q = m.begin(); q != m.end(); q++) {
  std::cout << q->first <<":"<< q->second << std::endl;
}
```

- The loop at the end outputs the entries in alphabetical order.
  - First **`Bob:17`**. Then **`Carlos:25`**. Then **`Gwen:50`**.

USE OF STD::UNORDERED_MAP IN STATS

- Below is a solution of Project 1's "stats", but using the STL:

```
// Build a dictionary of word counts using the text entered.
std::unordered_map<std::string,int> d { };

// Read until the end of text entry.
while (std::cin) {
  // Get the next line of entered text.
  std::string line;
  std::getline(std::cin,line);

  // Read each of the words in that line of text.
  for ( ... /* each word w in the line */ ...) {
    auto handle = d.find(w);
    if (handle == d.end()) {
      d[w] = 1;  // not found; create the entry
    } else {
      d[w]++;    // found; increment it
    }
  }
}
```

USE OF STD::UNORDERED_MAP/VECTOR/ALGORITHM IN STATS

▸(**STL_stats.cc** continued)

```
int numWords = d.size();
std::cout << "I saw " << numWords;
std::cout << " distinct words." << std::endl;

std::vector<std::pair<std::string,int>> words { };
for (auto p = d.begin(); p != d.end(); p++) {
  words.push_back(*p);
}

auto compare = [](std::pair<std::string,int> entry1,
                  std::pair<std::string,int> entry2) -> bool
               { return entry1.second > entry2.second; };
std::sort(words.begin(),words.end(),compare);

std::cout << "The 100 most-used words are:" << std::endl;
for (int i=0; i < 99; i++) {
  std::cout << words[i].first << ":" << words[i].second << ", ";
}
std::cout << words[99].first << ":" << words[99].second << std::endl;
```

USE OF STD::UNORDERED_MAP/VECTOR/LAMBDA ALGORITHM IN STATS

- (stats.cc continued)

This is a lambda expression!

```
int numWords = d.size();
std::cout << "I saw " << numWords;
std::cout << " distinct words." << std::endl;

std::vector<std::pair<std::string,int>> words { };
for (auto p = d.begin(); p != d.end(); p++) {
  words.push_back(*p);
}

auto compare = [](std::pair<std::string,int> entry1,
                  std::pair<std::string,int> entry2) -> bool
               { return entry1.second > entry2.second; };
std::sort(words.begin(),words.end(),compare);

std::cout << "The 100 most-used words are:" << std::endl;
for (int i=0; i < 99; i++) {
  std::cout << words[i].first << ":" << words[i].second << ", ";
}
std::cout << words[99].first << ":" << words[99].second << std::endl;
```

SUMMARY

- There are a ton of useful software components available in the C++ STL.
- Many of the common data structures: sequences, stacks, queues, dictionaries.
- Several useful algorithms, including sorting.
- The C++ template mechanism makes them widely applicable.
- Use them if you continue coding in C++ after this course!
- Lots of resources/tutorials on-line!
- What C++ remains: **copying/moving**, **lambda**, and **smart pointers**.

RECALL: USED A FUNCTION OBJECT IN SORT

```
// Build a list of word/frequency pairs from a dictionary.
std::vector<std::pair<std::string,int>> ws { };
for (auto p = d.begin(); p != d.end(); p++) {
  ws.push_back(*p);
}

// Sort in order of decreasing frequency.
auto compare = [](std::pair<std::string,int> entry1,
                  std::pair<std::string,int> entry2) -> bool
               { return entry1.second > entry2.second; };
//
std::sort(ws.begin(), ws.end(), compare);

// Output the top 100.
std::cout << "The 100 most-used words are:" << std::endl;
for (int i=0; i < 100; i++) {
  std::cout << ws[i].first << ":" << ws[i].second << std::endl;
}
```

C++ SYNTAX FOR A FUNCTION OBJECT

Here is the C++ syntax for an "anonymous" function object:

`[ ] (`*parameters*`) -> `*result-type*` { `*body*` }`

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

PYTHON SYNTAX FOR A FUNCTION OBJECT

Here is the Python syntax for an "anonymous" function object:

`(lambda` *parameters*`:` *expression* `)`

Examples:

```
(lambda n: n+1)                    // successor
(lambda x: x*x)                    // square
(lambda tens,ones: 10*tens+ones) // two_digit
```

C++ SYNTAX FOR A FUNCTION OBJECT

Here is the C++ syntax for an "anonymous" function object:

`[ ] (`*parameters*`) -> `*result-type*` { `*body*` }`

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) –> *result* { *rule* }

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[***captures*** **]** **(** ***parameters*** **)** **–>** ***result*** **{** ***rule*** **}**

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule. We'll look at this soon...
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[***captures*** **]** **(** ***parameters*** **)** **->** ***result*** **{** ***rule*** **}**

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) -> *result* { *rule* }

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) –> *result* { *rule* }

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

SYNTAX FOR A FUNCTION OBJECT

Here is the syntax for an anonymous function object:

[*captures*] (*parameters*) –> *result* { *rule* }

- ***captures***: list of variables from the context that are used *by value* or *by reference* in the rule.
- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](int n) -> int { return n+1; }                          // successor
[](double x) -> double { return x*x; }                    // square
[](int tens,int ones) -> int { return 10*tens+ones; } // two_digit
```

NOTE: can span multiple lines.

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

HERE ARE SOME USES

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -
                                     std::cout << n << s
                                     std::cout << n << s
                                     std::cout << n << s
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

BTW, this outputs:

```
1
11
5
54321
54321
54321
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

[***captures*** **] (** ***parameters*** **) –> *result* { *rule* }**

- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

`[` ***captures*** `]` `(` ***parameters*** `)` `->` ***result*** `{` ***rule*** `}`

- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

[***captures*** **] (** ***parameters*** **) –> *result* { *rule* }**

- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

EXAMPLE: COMPARISON FUNCTION OBJECT FOR SORT

Here is the syntax for an anonymous function object:

[***captures*** **] (** ***parameters*** **) –> *result* { *rule* }**

- ***parameters***: the function's parameters and their types
- ***result***: type of the returned result
- ***rule***: code that computes the return result from the parameters; the "body"

Examples:

```
[](std::pair<std::string,int> entry1,
   std::pair<std::string,int> entry2) -> bool
                { return entry1.second > entry2.second; }
```

NOTICE THE TYPES

```
std::function<bool(int)> isEven = [](int i) -> bool
                           { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                           { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                          { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                             std::cout << n << std::endl;
                             std::cout << n << std::endl;
                             std::cout << n << std::endl;
                           };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

NOTICE THE TYPES

At the top of this code:

```
#include <functional>
```

```
std::function<bool(int)> isEven = [](int i) -> bool
                                  { return (i % 2) == 0; };

std::function<int(int)> successor = [](int n) -> int
                                    { return n+1; };

std::function<void(void)> print5 = [](void) -> void
                                   { std::cout << 5 << std::endl; };

std::function<void(int)> print3x = [](int n) -> void {
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                     std::cout << n << std::endl;
                                   };
std::cout << isEven(10) << std::cout;
std::cout << successor(10) << std::cout;
print5();
print3x(54321);
```

SYNTAX FOR A FUNCTION OBJECT'S TYPE

Here is the syntax for an anonymous function object's type:

`std::function<` ***result-type*** `(` ***types-of-parameters*** `)>`

- ***types-of-parameters***: the function's parameter types
- ***result-type***: type of the returned result
- **Example:**

```
std::function<void(bool,std::string)> maybePrint =
  [](bool yes, std::string message) -> void {
    if (yes) {
      std::cout << message << std::endl;
    };
std::function<int(int,int)> two_digit =
  [](int tens_digit, int ones_digit) -> {
    return tens_digit*10 + ones_digit;
  }
```

CAPTURE BY VALUE

- Lambdas are defined *within the context* of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can *indicate that these things can be accessed* by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE

- Lambdas are defined *within the context* of executable code.
  - stack variables are available, stack objects are available
  - pointers to heap objects are available
- Can *indicate that these things can be accessed* by the function object.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
std::cout << make_fifty_something(8) << std::endl;
std::cout << make_fifty_something(7) << std::endl;
```

This outputs:

```
58
57
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

CAPTURE BY VALUE (CONT'D)

- NOTE: the variable is copied *by value*.
  - ➡ Subsequent changes aren't reflected because of this kind of capture.

Example:

```
int tens_digit = 5;
std::function<int(int)> make_fifty_something =
  [tens_digit](int ones_digit) -> int {
     return tens_digit * 10 + ones_digit;
  };
tens_digit--;
std::cout << make_fifty_something(8) << std::endl;
tens_digit--;
std::cout << make_fifty_something(7) << std::endl;
```

Also outputs:

```
58
57
```

FUN WITH VALUE CAPTURE

- I can invent "higher-order functions" that return functions as values.

Example:

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

```
std::function<void()> makePrinter(int x) {
  return [x]() { std::cout << x << "\n"; };
}

int main(void) {
  std::function<int(int)> add10 = makeAdder(10);
  std::cout << "add10(5) = " << add10(5) << std::endl;

  std::function<void()> print5 = makePrinter(5);
  std::cout << "print5(): " << std::endl;
  print5();
}
```

CAPTURE BY REFERENCE

- Alternatively, you can *indicate that variables' values can be tracked and altered* by the function object.
- You indicate this with the by-reference annotation **&**.

Example:

```
int tens_varies = 5;
std::function<int(int)> make_umpty_something =
  [&tens_varies](int ones_digit) -> int {
     return tens_varies * 10 + ones_digit;
  };
tens_varies--;
std::cout << make_umpty_something(8) << std::endl;
tens_varies--;
std::cout << make_umpty_something(7) << std::endl;
```

CAPTURE BY REFERENCE

- Alternatively, you can *indicate that variables' values can be tracked and altered* by the function object.
- You indicate this with the by-reference annotation **&**.

Example:

```
int tens_varies = 5;
std::function<int(int)> make_umpty_something =
  [&tens_varies](int ones_digit) -> int {
     return tens_varies * 10 + ones_digit;
  };
tens_varies--;
std::cout << make_umpty_something(8) << std::endl;
tens_varies--;
std::cout << make_umpty_something(7) << std::endl;
```

Outputs:

```
48
37
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

HERE ARE SOME USES OF REFERENCE

```
int count = 0;
std::function<void(void)> increment =
      [&count](void) -> void { count++; }
int x = 10;
int y = 11;
std::function<void(void)> increment =
      [&x,&y](void) -> void {
  int tmp = x;
  x = y;
  y = tmp;
};
increment();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
increment();
swap();
std::cout << count << " " << x << " " << y << std::endl;
```

This outputs:

```
1 10 11
2 11 10
3 10 11
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

HERE ARE SOME USES OF REFERENCE (CONT'D)

```
std::vector<int> v = {12, 8, 17};
std::function<void(void)> output =
  [&v](void) -> void {
    for (int e : v) {
      std::cout << e << " ";
    }
    std::cout << std::endl;
  };
std::function<void(int,int)> change =
   [&v](int index, int value) -> void { v.at(index) = value; };

output();
change(2,37);
output();
change(0,32);
output();
change(1,3);
output();
```

This outputs:

```
12 8 17
12 8 37
32 8 37
32 3 37
```

USE WITHIN ALGORITHMS PACKAGE

```
std::vector<int> v {0,1,4,9,16,25,36,49,64};
int sum = 0;
std::for_each(v.begin(),v.end(),
              [&sum](int e) -> void { sum +=e; } );
std::cout << "sum = " << sum << std::endl;
// outputs sum = 204
```

CAPTURE LIST

RECALL the syntax for an anonymous function object:

`[` ***capture-list*** `]` `(` ***parameters*** `)` `->` ***result*** `{` ***rule*** `}`

- ***capture-list***: list of variables from the context that are used in the rule
- Can be used *by value* or *by reference*:
  - If you want the variable's value to be copied use no annotation
  - If you want the stack object/variable to be referenced, changed, use **&**

MUTABLE KEYWORD

You can also make variables that are captured *by value* changeable:

`[` ***capture-list*** `](` ***parameters*** `)` **`mutable ->`** ***result*** `{` ***body*** `}`

- The mutable lets the body change the new variables that are copied
- This demonstrates its effect:

```
int startAt = 100;
std::function<int(void)> dbl =
  [startAt](void) mutable -> int {
     startAt *= 2; return startAt;
  };
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
```

MUTABLE KEYWORD

You can also make variables that are captured *by value* changeable:

`[` ***capture-list*** `](` ***parameters*** `)` **`mutable ->`** ***result*** `{` ***body*** `}`

- The mutable lets the body change the new variables that are copies
- This demonstrates its effect:

```
int startAt = 100;
std::function<int(void)> dbl =
  [startAt](void) mutable -> int {
     startAt *= 2; return startAt;
  };
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
std::cout << dbl() << " " << startAt << std::endl;
```

This outputs:

```
200 100
400 100
800 100
```

- The captured copy of the variable makes the lambda (internally) *stateful*.

SUMMARY OF C++ LAMBDA EXPRESSIONS

- C++ allows you to concisely express "function objects".
  - ➡ They are essentially one-time class instances with **`operator()`** defined.
- They are called *lambdas*
  - ➡ From the programming language **Lisp**: **`(lambda (n) (+ n 1))`**
  - ➡ Lisp's John McCarthy took them from Alonzo Church's "lambda calculus"
- Because C++ has a complex object memory model, must specify *captures*
  - ➡ overloads **`&`** syntax and uses keyword **`mutable`** to specify behavior
- Useful for many components defined in the **`algorithm`** STL

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

The function object returned by **`makeAdder`**, shown just below...

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}
```

...is essentially an instance of this class

```
class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

WHAT C++ BUILDS

With these definitions:

```
std::function<int(int)> makeAdder(int dx) {
  return [dx](int x) { return x+dx; };
}

class Foo {
private:
  int toAdd;
public:
  Foo(int dx) : toAdd {dx} {}
  int operator()(int x) const { return x+toAdd; }
};
```

We can write this:

```
std::cout << "(Foo {10})(5) = " << (Foo {10})(5) << std::endl;
std::cout << "(makeAdder(10))(5) = " << (makeAdder(10))(5)
          << std::endl;
```