

Lecture 14

Generics and STL

CS211 – Fundamentals of Computer Programming II
Branden Ghen a – Winter 2022

Slides adapted from:
Jesse Tov (Northwestern), Hal Perkins (Washington)

Administrivia

- Homework 5 due tonight
- Homework 6 should be released tonight
- Project details will be released tonight or tomorrow
 - First part will be groups proposing a project idea

Today's Goals

- Introduce concept of generic functions/classes
 - How they are made
 - How we used them
- Discuss major use case for generics
 - C++ Standard Template Library
- Understand how iterators allow generic traversal of a container

Getting the code for today

- Download code in a zip file from here:
https://nu-cs211.github.io/cs211-files/lec/14_generics_stl.zip
- Extract code wherever
- Open with CLion
 - Make sure you open the folder with the CMakeLists.txt
 - Details on CLion in Lab05

Outline

- **Generics**
- Standard Template Library
- Iterators
- Homework 6 Overview

Overloading functions to support multiple types

- Suppose you want a function that can compare any two things
 - Implement for `int` and implement for `float`

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise  
int compare(const int& value1, const int& value2) {  
    if (value1 < value2){ return -1;}  
    if (value2 < value1){ return 1;}  
    return 0;  
}
```

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise  
int compare(const float& value1, const float& value2) {  
    if (value1 < value2){ return -1;}  
    if (value2 < value1){ return 1;}  
    return 0;  
}
```

We want to avoid duplicated code

- The two implementations of `compare()` are nearly identical
 - Seems wasteful
- What if we want to extend `compare()` for other things?
 - `char, short, long, string, Position, String_Holder, etc.`
 - Impossible to get everything...

“Generic” version of the function

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise  
int compare(const ???& value1, const ???& value2) {  
    if (value1 < value2) { return -1; }  
    if (value2 < value1) { return 1; }  
    return 0;  
}
```

- What we would prefer is one “generic” version of the function
 - Code will be independent of what the real type is
 - One implementation works for everything!
 - Condition here: must implement `operator<()`

C++ Generics

- C++ implements generics through a concept called “templates”
- A template is a function or class that accepts a type as a parameter
 - You write the function code once in a type-agnostic way
 - When you invoke the function or instantiate the class, you specify the type as an argument to it
- At compile time, the compiler will generate the “specialized” code from your template that uses the type provided
 - The template definition is NOT runnable code
 - The compiler creates runnable code given a concrete type
 - A little like macro substitution

Generic functions

- Template to `compare()` any two things

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise  
template <typename T>    // <...> can also be written <class T>  
int compare(const T& value1, const T& value2) {  
    if (value1 < value2) return -1;  
    if (value2 < value1) return 1;  
    return 0;  
}
```

Generic functions

- Template to `compare()` any two things

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise  
template <typename T> // <...> can also be written <class T>  
int compare(const T& value1, const T& value2) {  
    if (value1 < value2) return -1;  
    if (value2 < value1) return 1;  
    return 0;  
}
```

- Declares the following function a template
 - The “generic” type is called `T`

Generic functions

- Template to `compare()` any two things

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename T> // <...> can also be written <class T>
int compare(const T& value1, const T& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```

- Declares the following function a template
 - The “generic” type is called `T`
- Code inside the template can use `T` like a type

Generic functions

- Template to `compare()` any two things

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise
template <typename COMPARE_TYPE>
int compare(const COMPARE_TYPE& value1, const COMPARE_TYPE& value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}
```

- We didn't have to name the type `T`
 - Could name it anything we want
 - Named in all capital letters by convention

Using generic functions

generic_compare.cxx

- Actual type being used goes in angle brackets after function name
 - `compare<COMPARE_TYPE>`

```
int main() {
    std::cout << compare<int>(10, 20) << "\n";
    std::cout << compare<double>(50.5, 50.6) << "\n";
    std::cout << compare<std::string>("hello", "world") << "\n";
    return 0;
}
```

Using generic functions

- The compiler can sometimes guess the correct type for you based on the arguments provided
 - This is known as "type inference"
- Can occasionally lead to unexpected results though...

```
int main() {  
    std::cout << compare(10, 20) << "\n";           // OK  
    std::cout << compare(50.5, 50.6) << "\n";       // OK  
    std::cout << compare("hello", "world") << "\n"; // FAILS!  
    return 0;  
}
```

Using generic functions

- The compiler can also guess the correct type for you based on the arguments provided
 - This is known as "type inference"
- Can occasionally lead to unexpected results though...
 - Third example below ends up calling `compare<char*>()`

```
int main() {
    std::cout << compare(10, 20) << "\n";           // OK
    std::cout << compare(50.5, 50.6) << "\n";       // OK
    std::cout << compare("hello", "world") << "\n"; // FAILS!
    return 0;
}
```


Generic classes

- Templates are most commonly used for classes (similarly structs)
- Entire class definition is templated
 - Template type can be used in data member and member functions

Example of generic classes

- Let's create a class called Pair that holds two "things"
 - The things do NOT have to be the same type
 - Like a tuple in python, but limited to two
- Operations
 - Set the value of the first thing
 - Set the value of the second thing
 - Get the value of the first thing
 - Get the value of the second thing
 - Print the pair of things
- Useful for the ability to return two things at once from a function!

Live coding: implement pair

generic_pair-complete.cxx

- Operations
 - Set the value of the first thing
 - Set the value of the second thing
 - Get the value of the first thing
 - Get the value of the second thing
 - Print the pair of things

- Real Pair implementation available in the C++ `<utility>` library
 - <https://www.cplusplus.com/reference/utility/pair/pair/>

Dangers of templates

generic_pair_compare.cxx

- Doing tricky things with compilers results in tricky errors
- Compiler error when you misuse a generic function (usually unintentionally!) can get really bad
 - Example: try calling `compare()` with something invalid
- Working with templates in general gets complicated and messy
- Need to implement all template code inside headers
 - Needs to be imported into each C++ file that uses it so the generated definitions are available

Generics in GE211

- You've already been using them!
 - `Posn<int>`, `Posn<float>`, `Dims<int>`, etc.
- You know enough to understand the entire implementation of `Posn`
 - Take a look at it when you get a chance
 - <https://github.com/tov/ge211/blob/2d7d3a1bd762c3b6d6fac791b0da2fc6c2013d3c/include/ge211/geometry.hxx#L264>

Break + Question

- What syntax would you use to create a Pair where both the values are Posns that hold ints?

```
Pair<??> pair({0, 0}, {3, 3});
```

Break + Question

- What syntax would you use to create a Pair where both the values are Posns that hold ints?

```
Pair<Posn<int>, Posn<int>> pair({0, 0}, {3, 3});
```

Outline

- Generics
- **Standard Template Library**
- Iterators
- Homework 6 Overview

C++ Standard Library

- Four major pieces
 1. The entire C standard library
 2. C++ input/output stream library
 - `std::cin`, `std::cout`, etc.
 3. C++ Standard Template Library (STL)
 - Containers, iterators, algorithms, etc.
 4. Miscellaneous other stuff
 - Strings, exceptions, memory allocation, localization

STL Containers

- Standard Template Library
 - Contains various useful functionality created as templates!
 - Apply for any type you want
- A container is an object that stores a collection of other objects
 - Like arrays or linked lists
- We already covered one of these: `std::vector`

STL `std::list`

- <http://www.cplusplus.com/reference/list/list/>
- A generic doubly-linked list
 - Next pointers and previous pointers allow movement in either direction
 - Can be more or less efficient than `std::vector`
 - See CS214

STL `std::unordered_map`

- https://www.cplusplus.com/reference/unordered_map/unordered_map/
- Generic map from key to value
 - For any type of key and type of value
 - Can store a value by its key
 - Can retrieve a value by its key
 - Works just like a python `dict`

Live coding: unordered_map example

unordered_map_example.cxx

```
int main() {
    std::unordered_map<std::string, int> map;

    map["CS211"] = 159;
    map["CE346"] = 30;
    std::cout << "map at CS211 = "
               << map["CS211"]
               << "\n";

    return 0;
}
```

Other STL containers <https://www.cplusplus.com/reference/stl/>

- Map
 - Key->Value in sorted order by key
- Set
 - Ordered list of unique elements
- Unordered_set
 - Unique elements in no particular order
- Array
 - Fixed size list of elements (like vector, but not resizable)
- And various others
 - Stack, Queue, etc.

Outline

- Generics
- Standard Template Library
- **Iterators**
- Homework 6 Overview

How do we make algorithms work on generic containers?

- C++ provides various algorithms in its `<algorithm>` library
 - `find()`, `count()`, `sort()`
- How does it make those work on any container?
 - Algorithm needs to traverse the container. But each container is different
 - Vector:

```
for(i=0; i<vector.size(); i++) {  
    vector[i];  
}
```
 - List:

```
for(node* curr=head; curr!=NULL; curr=curr->next) {  
    curr.value;  
}
```


Iterators allow generic traversing of containers

- Concept:
 - Create an object that allows you to move through the container
 - Holds a reference to the original object
 - Understands how to move through that specific implementation
- Operations an iterator must support:
 - Construction
 - Getting the value at the current location (* dereference)
 - Moving to the next location in the container (++)
 - Comparison with another iterator (== or !=)
 - Usually get two iterators, start and end, and traverse start until at end

General iterator pattern

```
start_iterator = object.begin();  
end_iterator = object.end();  
  
while (start_iterator != end_iterator) {  
    value = *start_iterator; // get value  
    // do something useful with value  
    start_iterator++; // move to next location  
}
```

Iterators are modeled after pointers!

iterator_example.cxx

```
int array[5] = {1, 2, 3, 4, 5};

int* start_iterator = &(array[0]);
int* end_iterator = &(array[5]);

while (start_iterator != end_iterator) {
    int value = *start_iterator;
    std::cout << "Value: " << value << "\n";
    start_iterator++;
}
```

Same code but for std::vector

iterator_example.cxx

```
std::vector<int> vec{1, 2, 3, 4, 5};
```

```
auto start_iterator = vec.begin();
```

```
auto end_iterator = vec.end();
```

```
while (start_iterator != end_iterator) {  
    int value = *start_iterator;  
    std::cout << "Value: " << value << "\n";  
    start_iterator++;  
}
```

auto asks the compiler to figure out the type for you

This part didn't have to change at all!

More complicated iterators can support more operations

- Depending on the container, iterators could support many operations
- Forward:
 - construction, equality, increment, get value
- Bidirectional:
 - Everything Forward does, decrement
- Random Access:
 - Everything Bidirectional does, arithmetic, comparison, get value at index

Live coding: use the count algorithm

iterator_example.cxx

- ```
int count(InputIterator first,
 InputIterator second,
 constT& value)
```

  - Counts occurrences of a value in a container
  - Actually returns an `iterator::difference_type`, but we'll ignore that
- We can count the number of times a certain value occurs inside a vector or array

# Break + Question

- How would we implement the following code?

```
int array[5] = {1, 1, 1, 2, 2};
```

```
// count the number of twos in array
```

```
int num_twos = count(???, ???, 2);
```

# Break + Question

- How would we implement the following code?
  - Pointers!

```
int array[5] = {1, 1, 1, 2, 2};
```

```
// count the number of twos in array
```

```
int num_twos = count(&(array[0]), &(array[5]), 2);
```

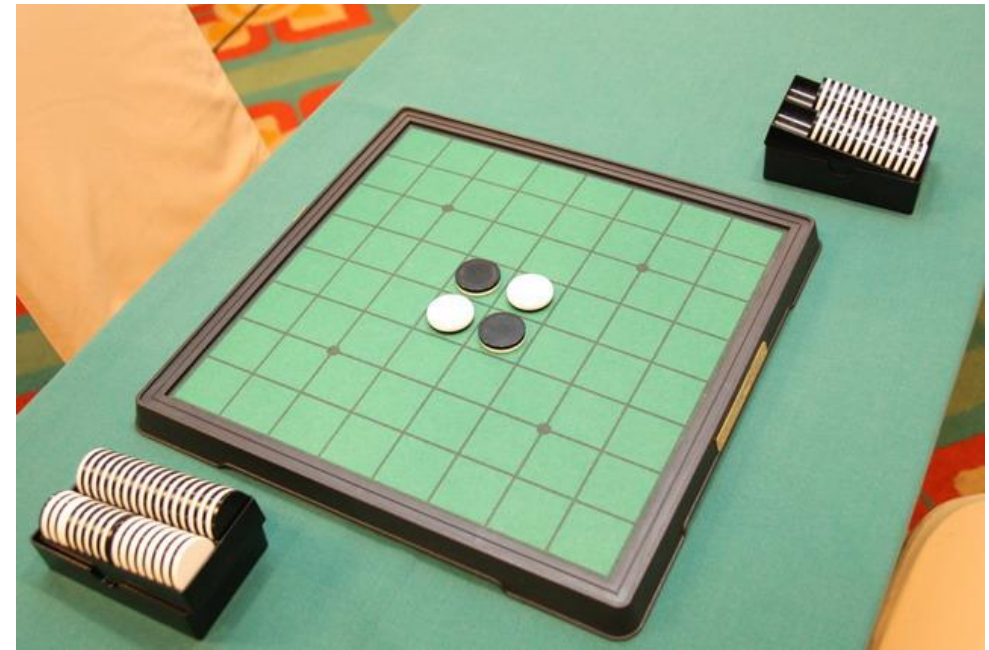


# Outline

- Generics
- Standard Template Library
- Iterators
- **Homework 6 Overview**

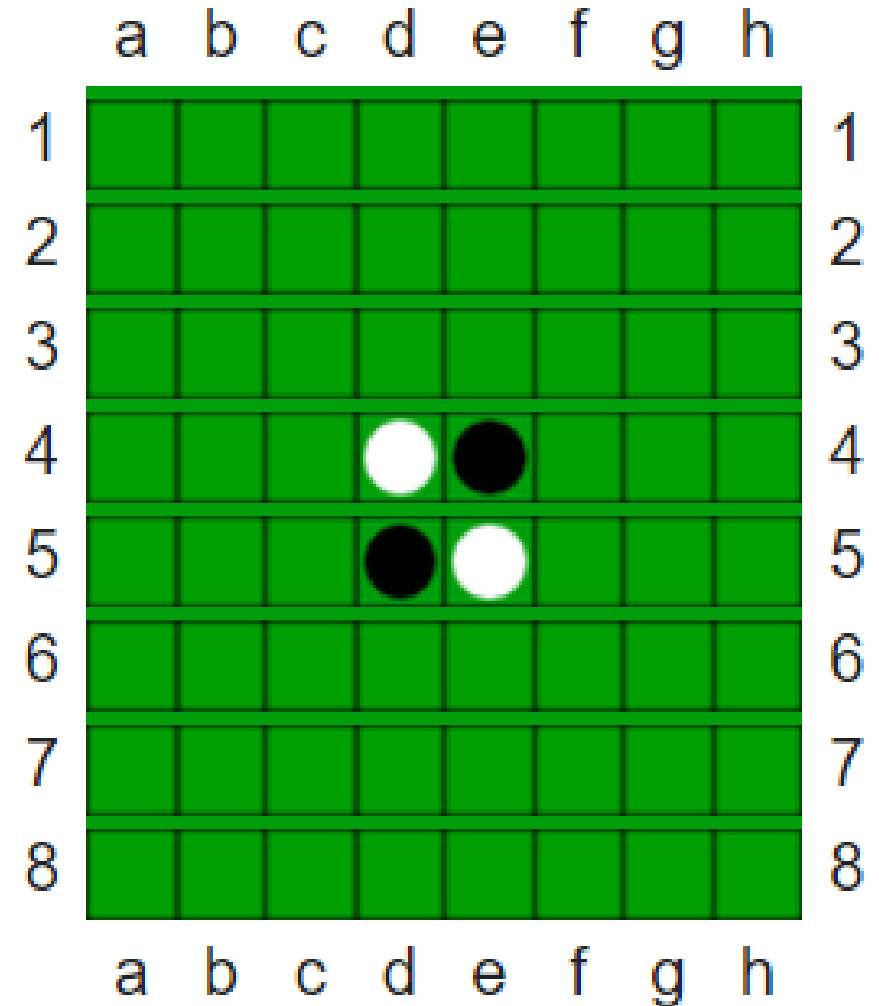
# Reversi

- Also known as Othello
- Light player and dark player take turns placing pieces
- A valid placed piece must be in a line with any number of opposing pieces followed by one piece of the current player
  - All opposing pieces in that bounded line are flipped to belong to the current player



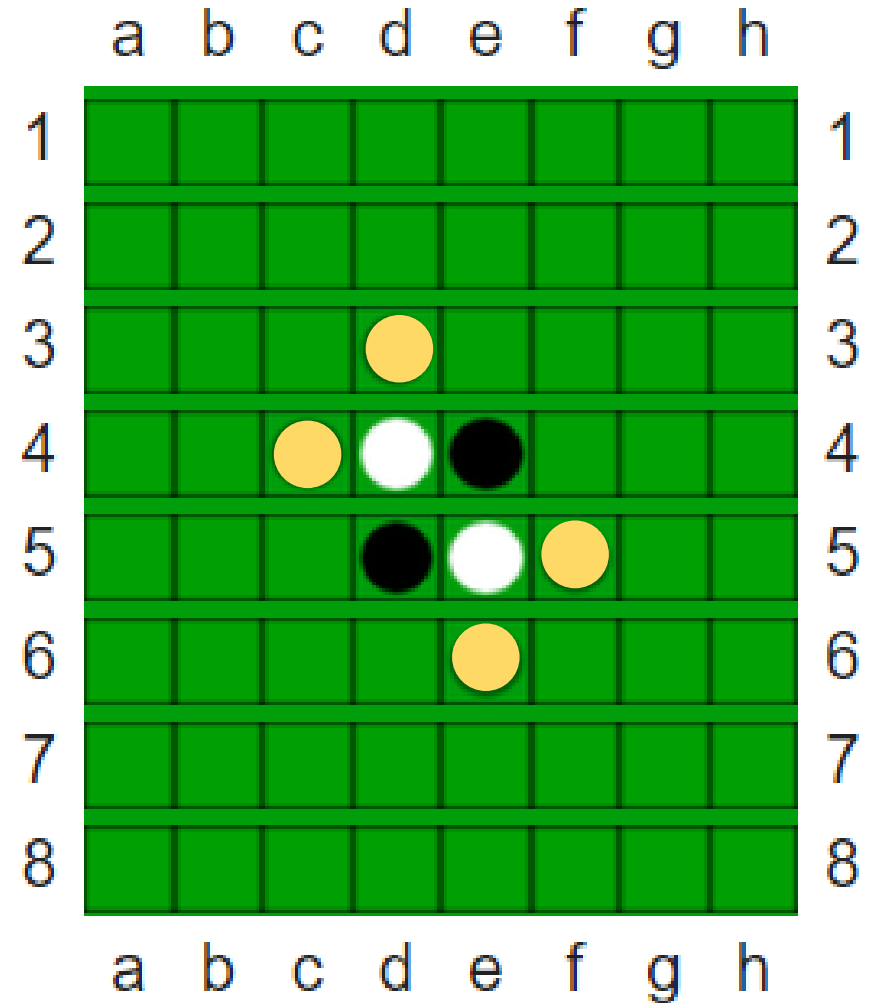
# Example move in reversi

- First, must place pieces in the central four squares
  - These don't follow the normal rules



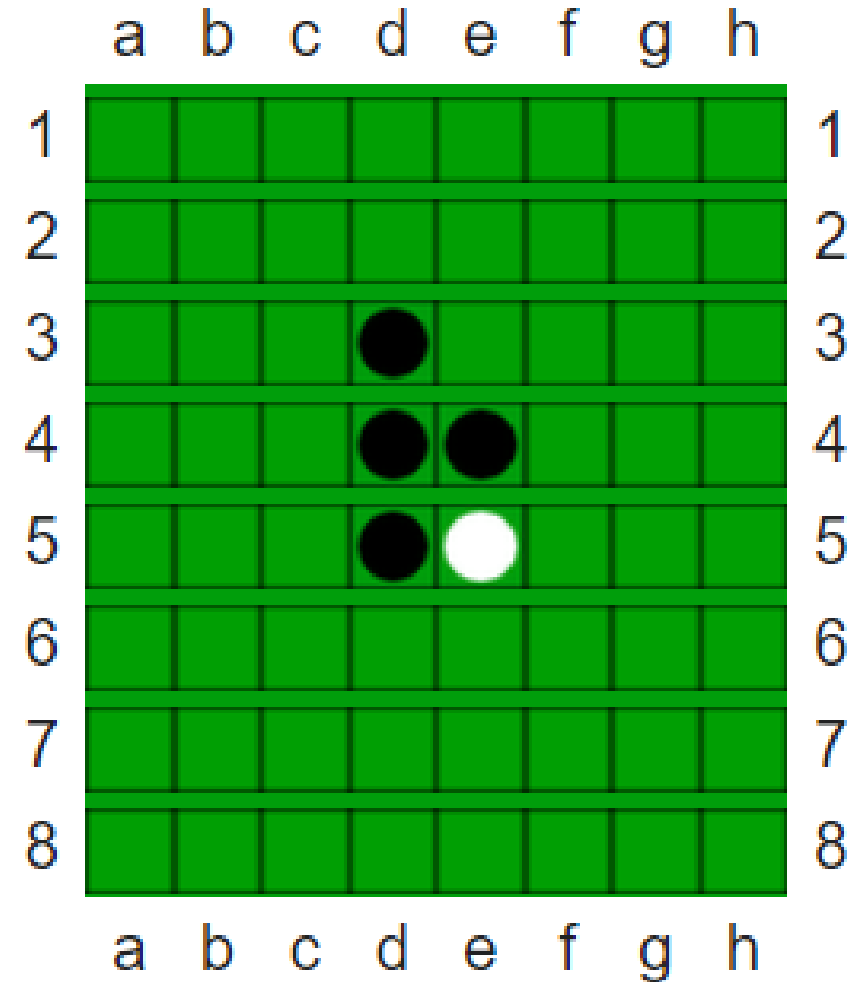
# Example move in reversi

- It is the dark player's turn
- They may play in any of the four locations indicated
  - Must form a line with a light piece in the middle



# Example move in reversi

- Once the dark player places a piece, all opposing pieces in that line are flipped



# Game demo

- <https://www.mathsisfun.com/games/reversi.html>
- Warning: the game setup rules are slightly different from ours
  - We let players play out the first two moves, which must be in the center

# Project layout

- Model, View, Controller
  - Same as with homework 5
  - **View** is responsible for drawing things
  - **Controller** gets inputs from the user
  - **Model** contains the game logic
- Model interacts with several other components
  - Board
  - Player
  - Move
    - Position\_set
    - Move\_map

# Player

- Represents a Player
  - Either in terms who owns a piece
  - Or whose turn it currently is

```
enum class Player {
 dark,
 light,
 neither
};
```



# Enums

- Define a new type with a fixed list of possible values

```
enum class Player {
 dark,
 light,
 neither
};
```

- New type: Player
  - Possible values: Player::dark, Player::light, Player::neither
- 
- Enums are in C as well as many other languages!

# Board

- Stores state for the game
  - Each `Posn<int>` within the board contains a `Player`
    - `Player::light`, `Player::dark`, or `Player::neither`
  - Valid positions are the rows/columns on the board
    - An 8x8 board goes from `{0,0}` to `{7,7}`
  - Can ask the board which piece is in a certain position
  - Can tell the board to set a piece in a certain position

# Move

- A `std::pair` of:
  - A position on the board
  - All pieces that would flip if the current player played in that position
    - Stored as a `Position_set`
- `Move_map`
  - An `std::unordered_map`
    - Holds Moves
  - Key is a position on the board
  - Value is the corresponding position set for the Move

# What do you have to do?

- Interact with a big program with lots of library files you didn't write
  - Board, Move, Player, Position\_set
  - You don't need to understand all of the code, but you do need to understand how to use them
    - Look through the .hxx files for them
- Fill the `Move_map` `next_moves`
  - Contents are each valid Move that the current player could make
  - Need to analyze the board to make that determination
- Eventually, you'll fill in the controller/view too
  - Including hints to the current player about possible places they could play

# Outline

- Generics
- Standard Template Library
- Iterators
- Homework 6 Overview