

# **Generics and The STL**

CMPT 125

Mar. 13

# Lecture 25

Today:

- Re-using Code Using Generics
- C++ `template`
- C++ Standard Template Library (STL)
  - `queue<T>`
  - `sort()`

# Code Re-Use (Review)

If a piece of code can be employed for multiple purposes, then you *factor* the code

- Principle: Write it once, and then re-use it.

We built a Queue ADT for integers. Can we easily re-use the code for:

- doubles?
- strings?
- ordered pairs?

Use C++ `template`

# Generic Programming

Express the algorithms so that they work on *any* type, to be specified as a parameter.

C++ uses the `template` construct to do this.

```
class queue {  
    private:  
        LL_t * intlist;  
    public:  
        queue();  
        ~queue();  
        int isEmpty();  
        void enqueue(int data);  
        int dequeue();  
};
```

```
template <class T>  
class queue {  
    private:  
        LL_t * intlist;  
    public:  
        queue();  
        ~queue();  
        int isEmpty();  
        void enqueue(int data);  
        int dequeue();  
};
```

# Generic Programming

Express the algorithms so that they work on *any* type, to be specified as a parameter.

C++ uses the `template` construct to do this.

```
class queue {  
    private:  
        LL_t * intlist;  
    public:  
        queue();  
        ~queue();  
        int isEmpty();  
        void enqueue(int data);  
        int dequeue();  
};
```

```
template <class T>  
class queue {  
    private:  
        Linked List of Ts;  
    public:  
        queue();  
        ~queue();  
        int isEmpty();  
        void enqueue(T data);  
        T dequeue();  
};
```

# Implementing Generic Methods

- Implement all methods in the header file.
- Prefix is: `template <class T> class<T>::`
- The typename `T` will be substituted throughout.

```
int queue::isEmpty() {  
    return list->isEmpty();  
}  
  
void queue::enqueue(int data) {  
    list->append(data);  
}  
  
int queue::dequeue() {  
    return list->removeHead();  
}
```

```
int queue::isEmpty() {  
    return list->isEmpty();  
}
```

# Implementing Generic Methods

- Implement all methods in the header file.
- Prefix is: `template <class T> class<T>::`
- The typename `T` will be substituted throughout.

```
int queue::isEmpty() {  
    return list->isEmpty();  
}  
  
void queue::enqueue(int data) {  
    list->append(data);  
}  
  
int queue::dequeue() {  
    return list->removeHead();  
}
```

```
template <class T>  
int queue<T>::isEmpty() {  
    return list->isEmpty();  
}  
  
void queue::enqueue(int data) {  
    list->append(data);  
}
```

# Implementing Generic Methods

- Implement all methods in the header file.
- Prefix is: `template <class T> class<T>::`
- The typename `T` will be substituted throughout.

```
int queue::isEmpty() {  
    return list->isEmpty();  
}  
  
void queue::enqueue(int data) {  
    list->append(data);  
}  
  
int queue::dequeue() {  
    return list->removeHead();  
}
```

```
template <class T>  
int queue<T>::isEmpty() {  
    return list->isEmpty();  
}  
  
template <class T>  
void queue<T>::enqueue(T data) {  
    list->append(data);  
}  
  
int queue::dequeue() {  
    return list->removeHead();  
}
```



# Implementing Generic Methods

- Implement all methods in the header file.
- Prefix is: `template <class T> class<T>::`
- The typename `T` will be substituted throughout.

```
int queue::isEmpty() {  
    return list->isEmpty();  
}  
  
void queue::enqueue(int data) {  
    list->append(data);  
}  
  
int queue::dequeue() {  
    return list->removeHead();  
}
```

```
template <class T>  
int queue<T>::isEmpty() {  
    return list->isEmpty();  
}  
  
template <class T>  
void queue<T>::enqueue(T data) {  
    list->append(data);  
}  
  
template <class T>  
int queue<T>::dequeue() {  
    return list->removeHead();  
}
```

# Implementing Generic Methods

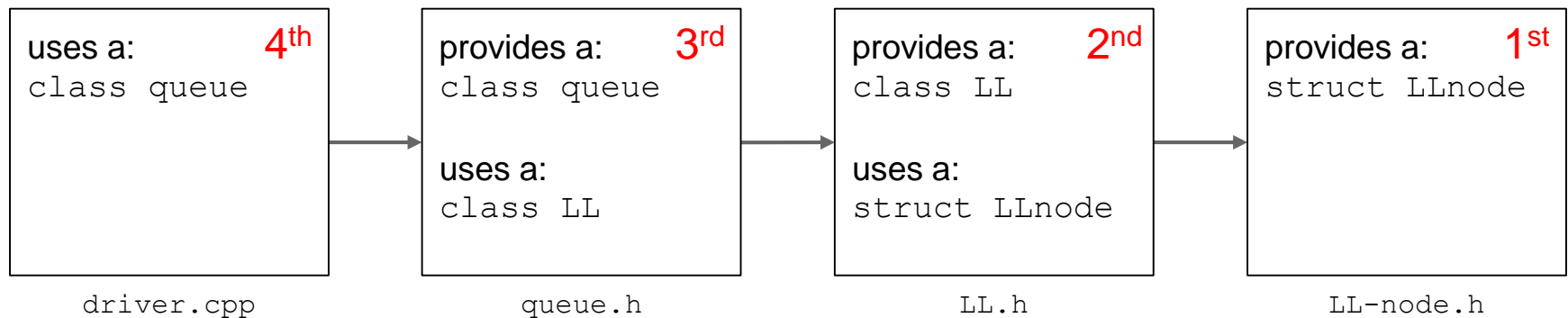
- Implement all methods in the header file.
- Prefix is: `template <class T> class<T>::`
- The typename `T` will be substituted throughout.

```
int queue::isEmpty() {  
    return list->isEmpty();  
}  
  
void queue::enqueue(int data) {  
    list->append(data);  
}  
  
int queue::dequeue() {  
    return list->removeHead();  
}
```

```
template <class T>  
int queue<T>::isEmpty() {  
    return list->isEmpty();  
}  
  
template <class T>  
void queue<T>::enqueue(T data) {  
    list->append(data);  
}  
  
template <class T>  
T queue<T>::dequeue() {  
    return list->removeHead();  
}
```

# Example: Developing Generics

In our Queue ADT example, there was dependency among the code.



*A dependency graph* shows the dependency relationship among entities (files and classes)

- $X \rightarrow Y$  means that X depends on Y

Strategy: start at the end without dependencies

# Compiling Your Code In Multiple Files

- Before, to compile C code, you used gcc
- Now, to compile C++ code, use g++
- Compile all files with automatically chosen output file (a.out)

```
g++ LL-node.h LL.cpp LL.h driver.cpp queue.cpp queue.h
```

- Compile all files but choose output file name (main)

```
g++ -o main LL-node.h LL.cpp LL.h driver.cpp queue.cpp queue.h
```

# Some Useful g++ Options

- `-Wall`: turn most warnings on
- `-o <name>`: name of output file
- `-c`: output an object file (.o), used for multi-stage compilation
- `-I<include path>`: specify an include directory
- `-L<include path>`: specify an library directory
- `-g`: turn on debugging

# Makefile

- A linux utility called “make” will look for a file called “makefile” in the same directory
- A makefile makes the compilation process more convenient
  - Only compiles files that have changed

# Makefile

- Makefile contains blocks of instructions for compiling the code.

```
target: prerequisites
```

```
    command
```

```
    command
```

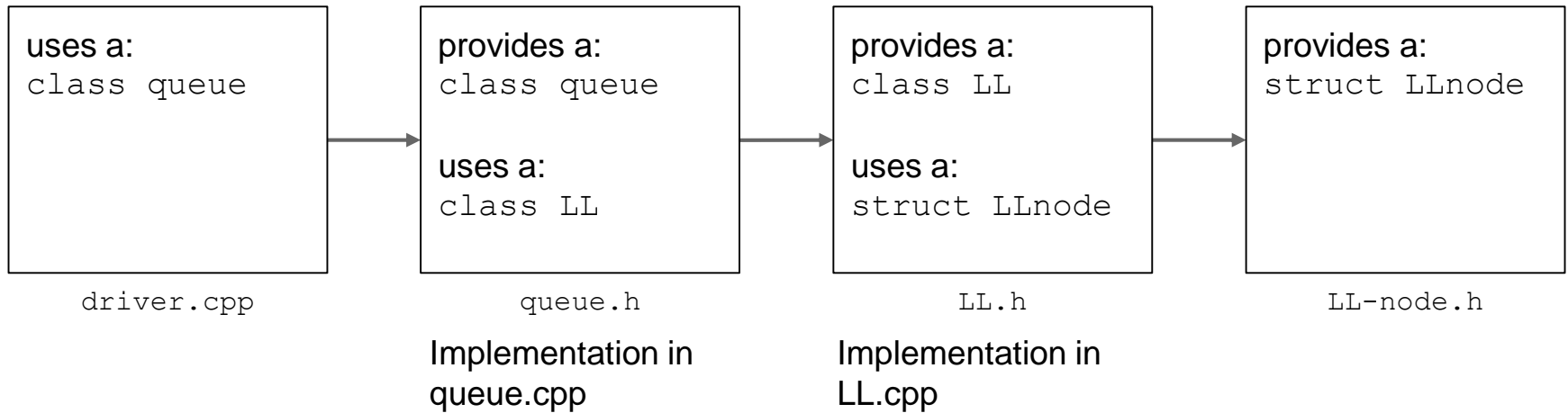
```
    ...
```

tab is required!

- Each block contains three parts
  - “target”: name of executable or object file generated by g++, or name of action(s) to carry out
  - “prerequisites”: a list of files that are needed to create/perform the target (AKA dependencies)
  - “command”: the actual action to be carried out, in the form of a g++ command

# Makefile

## Our code from last class



- Compile LL and LL-node

```
LL.o: LL.h LL.cpp LL-node.h  
g++ -Wall -c LL.cpp
```

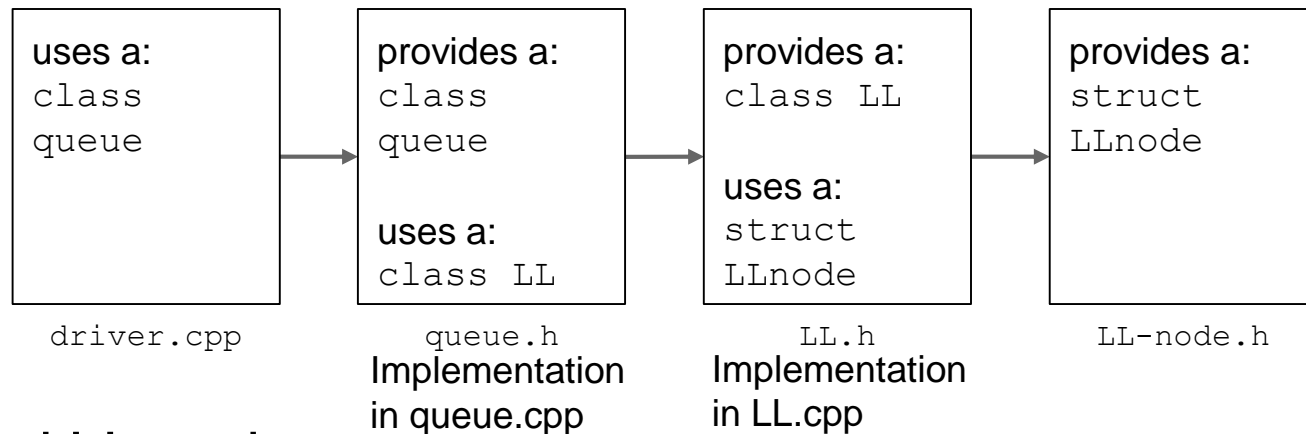
prerequisites

target (output file)

command (just 1 command in this case)



# Makefile



- Compile LL and LL-node

```
LL.o: LL.h LL.cpp LL-node.h  
g++ -Wall -c LL.cpp
```

- Compile queue

```
queue.o: LL.h queue.cpp queue.h  
g++ -Wall -c queue.cpp
```

- Compile driver

```
driver.o: driver.cpp queue.h  
g++ -Wall -c driver.cpp
```

- Compile driver, which includes all above compilations

```
driver: driver.o LL.o queue.o  
g++ -Wall -o driver driver.o queue.o LL.o
```

# Makefile

- Cleaning up
- You always need a target called “clean”, that allows users to remove all compiled files
  - This allows us to share source code, and let the other person compile, instead of sharing the application

```
clean:
```

```
    rm -f driver *.o *.gch
```

# Pitfalls with Generics

Some commands are not easily made generic

- E.g., `return -1;` in method `removeHead()`
- E.g., `printf( . . . )` in method `print()`

C++ offers some solutions:

- operator overloading
- streams
- exceptions

All of these are possible, but not wonderful

- . . . and not recommended
- . . . and beyond the scope of CMPT 125 / 118

# C++ Standard Template Library (STL)

A collection of containers, definitions and algorithms for common use.

```
#include <queue>

int main () {
    std::queue<int> *Q = new std::queue<int>;
    for (int i = 0; i < 5; i++)
        Q->push(i);
    while (!Q->empty()) {
        printf("front = %d, back = %d\n", Q->front(), Q->back());
        Q->pop();
    }
    delete Q;
    return 0;
}
```

## STL `queue<T>` Interface:

- `empty()` — same as `isEmpty()`
- `push(x)` — same as `enqueue(x)`
- `pop()` — similar to `dequeue()`, except no return value
- `front()` — returns a copy of the next item to be dequeued
- `back()` — returns a copy of the last item that was enqueued

# C++ Standard Template Library (STL)

A collection of containers, definitions and algorithms for common use.

```
#include <queue>
```

```
using namespace std;
```

```
int main () {  
    queue<int> *Q = new queue<int>;  
    for (int i = 0; i < 5; i++)  
        Q->push(i);  
    while (!Q->empty()) {  
        printf("front = %d, back = %d\n", Q->front(), Q->back());  
        Q->pop();  
    }  
    delete Q;  
    return 0;  
}
```

## STL queue<T> Interface:

- `empty()` — same as `isEmpty()`
- `push(x)` — same as `enqueue(x)`
- `pop()` — similar to `dequeue()`, except no return value
- `front()` — returns a copy of the next item to be dequeued
- `back()` — returns a copy of the last item that was enqueued

For more natural usage of the standard template library.

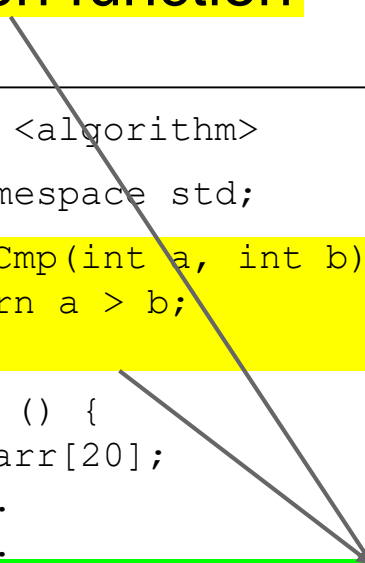
# Generic sort () in STL

There are two versions of `sort ()`:

- one that uses the built-in less than operator `<`
- one that uses your own **comparison function**

```
#include <algorithm>
using namespace std;
int main () {
    int arr[20];
    . . .
    sort(arr, arr+20);
    . . .
}
```

```
#include <algorithm>
using namespace std;
bool intCmp(int a, int b) {
    return a > b;
}
int main () {
    int arr[20];
    . . .
    sort(arr, arr+20, intCmp);
    . . .
}
```



Q. What's the difference between these?

**Look up `sort()` in STL**