CSE 1325: Object-Oriented Programming

Lecture

Templates

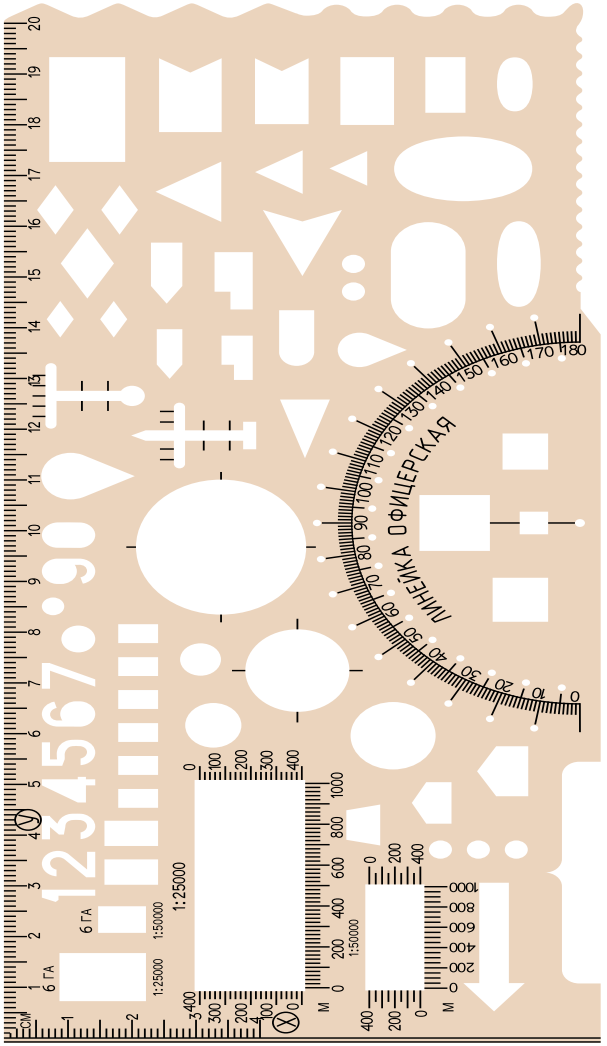
Mr. George F. Rice

george.rice@uta.edu

A truly happy person is one who can enjoy the scenery on a detour.

Overview: Templates

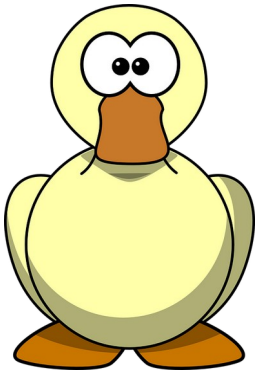
- Templates
  - Difference with Java generics
  - C++ Implementation
- Examples
  - Max (function)
  - Vector (class)
  - Get (method)
  - to_string and stox (function)
  - Double vector (class)



Template image is in the public domain.
Yes, we used these when I was in university!

Generic Programming in C++

- Like Java, C++ supports generic programming using *templates*
  - It's the same thing conceptually as a Java *generic*
  - But the implementations are radically different
- Differences with Java generics
  - **C++ *expands* the code**, replicating it once for each type instanced in your program and creating many new types
    - **Java *erases* the type**, so the *exact same binary instructions* are used
    - C++ can also have specialized template instances for optimization
  - **C++ templates work with primitive types** as well as class types
    - And **C++ permits default type arguments** as well
  - **C++ uses "duck typing"** instead of a constraint language
    - If it looks like a duck and quacks like a duck, it's a duck!
    - If it compiles, it's compatible! If not, it's not compatible!



Duck image by Clker-Free-Vector-Images per the Pixabay License.

C++ Templates

- Templates are “smart” preprocessor macros
  - We parameterize the types inside angle brackets
  - The *compiler* validates the types when the template is instanced into a class or function

```
template <class T>
T max(T lhs, T rhs) {
    if (lhs > rhs) return lhs; else return rhs;
}

int main() {
    std::cout << "The larger of 3 and 42 is " << max(3, 42) << std::endl;
    std::cout << "The larger of 3.1415 and 2.718 is " << max(3.1415, 2.718) << std::endl;
    std::cout << "The larger of (1,2,3) and (2,1,-4) is "
              << max(Coordinate{1,2,3},Coordinate{2,1,-4}) << std::endl;
}
```

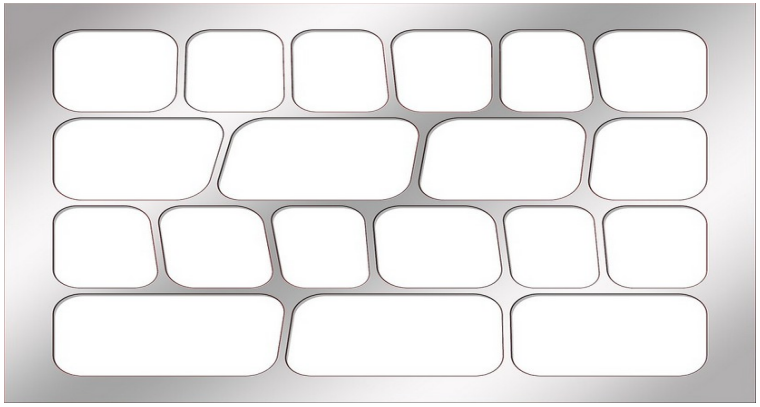
`operator<` must be defined for any type `T` used with function `max`

This keeps only one copy of the bugs
and allows the code to be easily readable and maintainable

Templates

- **Template** – A C++ construct representing a class, method, or function in terms of generic types
  - This enables the algorithm to be written independent of the types of data to which it applies
  - A type may be specified when the template class is instanced or the template method / function is called
    - Though this is not always necessary, as shown in the previous example

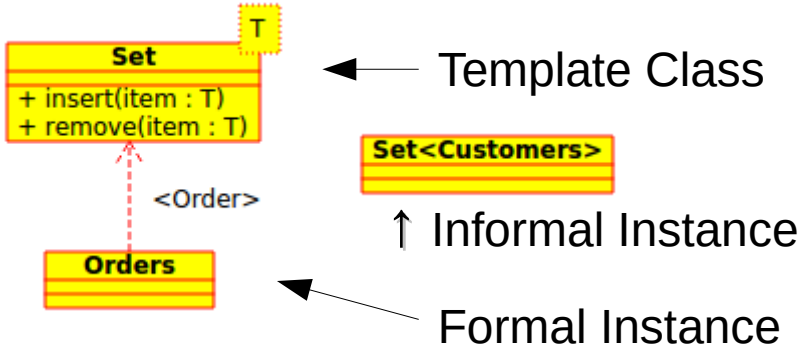
- As with dynamic languages, the type specified must supply the needed methods
  - For example, our max template will only work if the generic type overrides the > operator



Generic Programming

- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation
  - Called templates in C++, generics in Java, and parameterized types in the UML

UML Notation



Generic Class Example: A Time-Tagged ~~ArrayList~~ Vector

- Very similar to Java's TaggedArrayList except
  - We can access a TaggedObject via a subscript
- Use classic time_t struct
  - time(0) creates the current time and date
  - ctime(&date) formats the time_t into a char* (with a trailing newline that we must remove, seriously?)
- We will implement composition only
  - Believe it or not, you cannot extend std::vector!

Java Generics Review

TaggedObject Generic Class

- This was our Time-Tagged Array List in Java from Lecture 13

In ArrayListTimeTaggedGeneric.java

```
import java.util.Date;
import java.text.SimpleDateFormat;

class TaggedObject<E> {
    public TaggedObject(Date date, E value) {
        this.date = date;
        this.value = value;
    }
    public String toString() {
        return "'" + value + "' (at " + formatDate.format(date) + ")";
    }

    public Date date;
    public E value;
    private static SimpleDateFormat formatDate =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
}
```

Java Generics Review

TaggedArrayList Generic Class

- We implement only the minimum methods required for our testing – add, get, when, size
  - Other methods could be added as needed
  - No sync problems – every element MUST have a correct date

```
import java.util.ArrayList;
import java.util.List;

class TaggedArrayList<E> {
    public TaggedArrayList() {list = new ArrayList<>();}

    public void add(E element) {list.add(new TaggedObject<E>(new Date(), element));}
    public E get(int index) {return list.get(index).value;}
    public Date when(int index) {return list.get(index).date;}
    public int size() {return list.size();}

    public String toString(int index) {return list.get(index).toString();}
    private ArrayList<TaggedObject<E>> list;
}
```

Java Generics Review
A Quick Demo

- Store two strings and print them out with dates
- The enterprising student could readily write interactive test programs – right?
  - Enter strings at a prompt, store them, and list with dates at the end
  - Enter strings, then find a specified string and print its creation date

```
public class ArrayListTimeTaggedGeneric {
    public static void main(String[] args) throws InterruptedException {
        TaggedArrayList<String> list = new TaggedArrayList<>();
        System.out.print("Working...");

        list.add("The answer is ");
        Thread.sleep(1000 + (long)(Math.random() * 5000)); // wait 1-6 seconds
        list.add("forty-two");

        System.out.println("done!");

        for (int i=0; i<list.size(); ++i) {
            System.out.println(list.toString(i));
        }
    }
}
```

```
Working...done!
'The answer is ' (at 2023-04-11 22:04:12)
'forty-two' (at 2023-04-11 22:04:16)
```

C++ Template
TaggedObject

- The template declaration *and definition* must be in the .h file
  - Remember C++ *expands* the code – so it needs the code whenever the template is instanced
- The keyword template with <class T> (or <typename T>) precedes the class or function definition
  - Rather than inline as with Java

All code in the .h file!

tagged_vector.h

```
#include <iostream>
#include <sstream>
#include <vector>
#include <ctime>

// This class stores a time-tagged value of type T
template<class T>
class TaggedObject {
  public:
    TaggedObject(time_t date, T value) : _date{date}, _value{value} { }

    time_t _date;
    T _value;
};
```

C++ Template

TaggedObject-related Functions

- We'll also define a couple of template functions
  - operator<< for a TaggedObject<T>
  - to_string for ANY object
    - This uses an output string stream to convert the object using its operator<< definition, making operator<< the toString of C++

tagged_vector.h

```
template <class T>
std::ostream& operator<< (std::ostream& ost, const TaggedObject<T>& to) {
    std::string date = std::string{ctime(&to._date)};
    date.pop_back(); // ctime appends a newline (seriously???) - remove it
    ost << "'" << to._value << "' (at " << date << ")";
    return ost;
}

template <class T>
std::string to_string(const T& value) {
  std::ostringstream oss;
  oss << value;
  return oss.str();
}
```

This generic method works with ANY C++ type!

C++ Template
TaggedVector

- Now our TaggedVector is quite straightforward
  - Access the TaggedObject with subscripting: **`v[i]`**
  - Access the T value directly with get: **`v.get(i)`**
  - Access the time_t date with when: **`v.when(i)`**

tagged_vector.h

```
template <class T>
class TaggedVector {
  public:
    void push_back(T element) {v.push_back(TaggedObject(time(0), element));}

    TaggedObject<T>& operator[](int index) {return v[index];}
    T get(int index) {return v[index]._value;}
    time_t when(int index) {return v[index]._date;}

    int size() {return v.size();}
    std::string to_string(int index) {return to_string(v[index]);}

  private:
    std::vector<TaggedObject<T>> v;
};
```

C++ Template
A Quick Demo

- Store two strings and print them out with dates
- Output is very similar to Java

main.cpp

```cpp
#include "tagged_vector.h"

int main() {
    TaggedVector<std::string> v;
    v.push_back("The answer is ");
    v.push_back("42");
    for(int i=0; i<v.size(); ++i)
        std::cout << v[i] << std::endl;
}
```

C++

```
Working...done!
'The answer is ' (at Tue Apr 11 22:20:36 2023)
'42' (at Tue Apr 11 22:20:41 2023)
```

Java

```
Working...done!
'The answer is ' (at 2023-04-11 22:09:43)
'forty-two' (at 2023-04-11 22:09:48)
```

TaggedObject Class Comparison

Java

```java
class TaggedObject<E> {
    public TaggedObject(Date date, E value) {
        this.date = date;
        this.value = value;
    }
    public String toString() {
        return "'" + value + "' (at " + formatDate.format(date) + ")";
    }
    public Date date;
    public E value;
    private static SimpleDateFormat formatDate =
        new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
}
```

C++

```cpp
template<class T>
class TaggedObject {
  public:
    TaggedObject(time_t date, T value) : _date{date}, _value{value} { }

    time_t _date;
    T _value;
};
template <class T>
std::ostream& operator<< (std::ostream& ost, const TaggedObject<T>& to) {
    std::string date = std::string{ctime(&to._date)};
    date.pop_back(); // ctime appends a newline (seriously???) - remove it
    return ost << "'" << to._value << "' (at " << date << ")";
}
```

TaggedArrayList vs TaggedVector Class Comparison

Java

```java
class TaggedArrayList<E> {
    public void add(E element) {list.add(new TaggedObject<E>(new Date(), element));}

    public E get(int index) {return list.get(index).value;}
    public Date when(int index) {return list.get(index).date;}

    public int size() {return list.size();}
    public String toString(int index) {return list.get(index).toString();}

    private ArrayList<TaggedObject<E>> list = new ArrayList<>();
}
```

C++

```cpp
template <class T>
class TaggedVector {
  public:
    void push_back(T element) {v.push_back(TaggedObject(time(0), element));}

    TaggedObject<T>& operator[](int index) {return v[index];}
    T get(int index) {return v[index]._value;}
    time_t when(int index) {return v[index]._date;}

    int size() {return v.size();}
    std::string to_string(int index) {return to_string(v[index]);}

  private:
    std::vector<TaggedObject<T>> v;
};
```

Main Comparison

Java

```java
public class ArrayListTimeTaggedGeneric {
    public static void main(String[] args) throws InterruptedException {
        System.out.print("Working...");
        TaggedArrayList<String> list = new TaggedArrayList<>();
        list.add("The answer is ");
        Thread.sleep(1000 + (long)(Math.random() * 5000));
        list.add("forty-two");
        System.out.println("done!");
        // Haven't covered iterators yet, so 3-term for
        for (int i=0; i<list.size(); ++i) {
            System.out.println(list.toString(i));
        }
    }
}
```

C++

```cpp
#include "tagged_vector.h"

// Include sleep() function on Windows or Linux / Unix / Mac

int main() {
    TaggedVector<std::string> v;
    std::cout << "Working...";
    std::cout.flush();

    v.push_back("The answer is ");
    sleep(5);
    v.push_back("42");

    std::cout << "done!" << std::endl;
    for(int i=0; i<v.size(); ++i)
        std::cout << v[i] << std::endl;
}
```

Templates Live in the .h

- The rule has been “Declare in .h, Define in .cpp”
  - But templates are a different animal – they are all declarations *until they are instanced*
  - Once instanced, C++ begins actually generating code (using the parameterized type)
  - Thus you can safely put the entire definition of the template in the .h with no duplicate definition errors
- And you must!
  - Otherwise, C++ will not have the *definition* of the template from which to generate code when you instance the template! You get linker errors...

Templates Live in the .h

- Splitting a template between .h and .cpp results in linker errors like this:

```
test_lifo.o: In function `test_underflow()':
test_lifo.cpp:(.text+0x49c): undefined reference to `LIFO<double>::LIFO(int)'
test_lifo.cpp:(.text+0x4bc): undefined reference to `LIFO<double>::push(double const&)'
test_lifo.cpp:(.text+0x4c8): undefined reference to `LIFO<double>::pop()'
test_lifo.cpp:(.text+0x4d4): undefined reference to `LIFO<double>::pop()'
test_lifo.cpp:(.text+0x4fc): undefined reference to `LIFO<double>::~LIFO()'
test_lifo.cpp:(.text+0x52c): undefined reference to `LIFO<double>::~LIFO()'
```

Bottom Line

- For *normal code*, put declarations in .h and definitions in .cpp
- For *templates*, put declarations AND definitions in .h

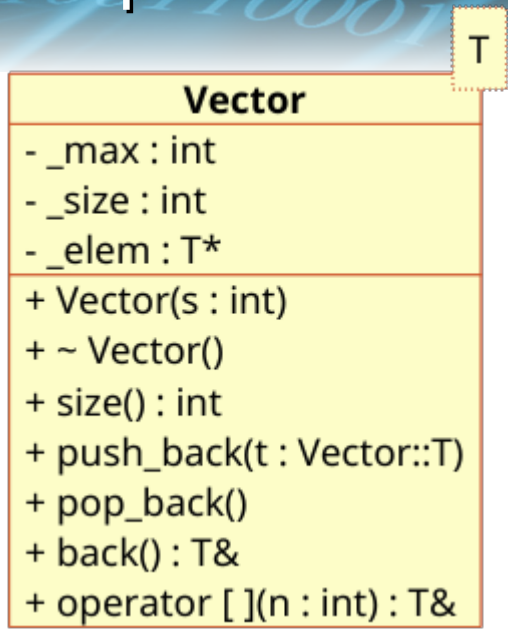
TaggedVector via Inheritance

- In Java, we implemented TaggedArrayList twice
  - Using composition, by including the ArrayList
  - Using inheritance, by extending the ArrayList
- We cannot do that with C++
  - **Vector cannot be extended!**
  - This is true of all of the Standard Template Library containers (what Java calls collections)
  - They cannot be extended!
  - So we use composition with STL members in C++

Another Template Example: Vector

Our Vector *can* be extended – but it's less capable!

- std::vector is a template
  - Can store any type of element
  - Provides same operations for all types
- std::vector was written by a person
  - Nothing magical about it!
- Let's write our own Vector implementation
  - How hard can it be?
  - Not that hard, actually – for a *simplified* version
    5 methods including all-important [ ] for indexing



UML Template Notation

Vector Template
Is that *all* the code?

- Compact coding – we range check with assert instead of properly throwing an out_of_range exception (*blush*)

vector.h

```
#ifndef __VECTOR_H
#define __VECTOR_H

#include <cassert>

template <class T>
class Vector {
  public:
    Vector(int s = 10) : _size{0}, _max(s), _elem{new T[s]} { }
    ~Vector() {delete[] _elem;}
    int size() {return _size;}
    void push_back(T t) {_elem[_size] = t; assert(_size++ < _max);}
    void pop_back() {assert(_size-- > 0);}
    T& back() {return operator[](_size-1);}
    T& operator[](int n) {assert(0 <= n && n < _size); return _elem[n];}
  private:
    int _max;  // size of the array
    int _size; // next array element
    T*  _elem; // a pointer to the elements
};

#endif
```

What's 2 key members are missing?

Vector Template

It is NOT *all* the code!

- Rule of 3! If you have a destructor, copy constructor, or copy assignment operator, you probably need all 3!

vector.h

```
#ifndef __VECTOR_H
#define __VECTOR_H

#include <cassert>

template <class T>
class Vector {
  public:
    Vector(int s = 10) : _size{0}, _max(s), _elem{new T[s]} { }
    ~Vector() {delete[] _elem;}
    int size() {return _size;}
    void push_back(T t) {_elem[_size] = t; assert(_size++ < _max);}
    void pop_back() {assert(_size-- > 0);}
    T& back() {return operator[](_size-1);}
    T& operator[](int n) {assert(0 <= n && n < _size); return _elem[n];}
  private:
    int _max;  // size of the array
    int _size; // next array element
    T*  _elem; // a pointer to the elements
};

#endif
```

Vector Template

Here's the *rest* of the code!

- Copy constructor & copy assignment operator
  - Allocate **`new`** array and then copy elements

vector.h

```
template <class T>
class Vector {
  public:
    Vector(int s = 10) : _size{0}, _max(s), _elem{new T[s]} { }
    //    --> Destructor - Free the array from heap memory
    ~Vector() {delete[] _elem;}
     //   --> Copy Constructor - Create a new array and copy the data
    Vector(const Vector<T>& v) {
        _size=v._size;
        _max=v._max;
        _elem = new T[_size];
        for (int i=0; i<_size; ++i) _elem[i] = v._elem[i];
    }
    //   --> Copy Assignment Operator - Create a new array and copy the data
    Vector<T>& operator= (const Vector<T>& v) {
        if(this == &v) return *this;
        delete[] _elem;
        _size = v._size;
        _max = v._max;
        _elem = new T[_size];
        for (int i=0; i<_size; ++i) _elem[i] = v._elem[i];
        return *this;
    }
```

Remaining code is unmodified

Regression Test for Vector Template
(1 of 2)

- Exercises each method we defined

test_vector.h

```
#include "vector.h"
#include <iostream>

int main() {
    const int SIZE = 10;
    Vector<int> vi;
    for (int i=0; i<SIZE; ++i)
        vi.push_back(i*i);

    if (vi.back() != 81)
        std::cerr << "FAIL: v.back() is " << vi.back()
                  << " instead of 81" << std::endl;

    for (int i=vi.size()-1; i>=0; --i) {
        if (vi[i] != i*i) {
            std::cerr << "FAIL: stored " << vi[i]
                      << " instead of " << i*i << std::endl;
        }
    }
    vi[5] = 42;
    if (vi[5] != 42)
        std::cerr << "FAIL: set vi[5] to 42 but is " << vi[5]
                  << " instead" << std::endl;

}
```

Regression Test for Vector Template
(2 of 2)

- Try it with class type as well!

test_vector.h

```
    // Test with string
    const int SIZES = 5;
    Vector<std::string> vs{SIZES};
    vs.push_back("Hello, world!");
    vs.push_back("Farewell, cruel world!");

    if(vs[0] != "Hello, world!")
        std::cerr << "FAIL: v[0] was " << vs[0]
                  << "instead of 'Hello, world!'" << std::endl;
    if(vs[1] != "Farewell, cruel world!")
        std::cerr << "FAIL: v[0] was " << vs[1]
                  << "instead of 'Farewell, cruel world!'" << std::endl;

    vs[0] = "Hang in there, world!";
    if(vs[0] != "Hang in there, world!")
        std::cerr << "FAIL: v[0] was " << vs[0]
                  << "instead of 'Hang in there, world!'" << std::endl;

}
```

```
ricegf@pluto:~/dev/cpp/cse1325-prof/21/code_from_slides/vector$ make clean ; make
rm -f *.o *.gch *~ test_vector
g++ --std=c++17 -c test_vector.cpp
g++ --std=c++17 -o test_vector test_vector.o
ricegf@pluto:~/dev/cpp/cse1325-prof/21/code_from_slides/vector$ ./test_vector
ricegf@pluto:~/dev/cpp/cse1325-prof/21/code_from_slides/vector$
```

A Simple Template Method

- A generic method in a non-generic class

generic_method.cpp

```cpp
#include <iostream>

class Test {
  public:
    template<class T>
    T get();
};

// Implementation (remember, must be in the HEADER file if you use one!)
template<class T>
T Test::get() {
    T t;
    return t;
}

// Using Test's generic method
int main() {
    Test t;
    std::cout << t.get<int>() << std::endl;
}
```

Class Test is NOT generic, but method get() is!

Since ints are not initialized to 0 in C++, we get whatever value was already in memory.

```
ricegf@antares:~/dev/202308/23-c++-templates/code_from_slides$ c17 generic_method.cpp
ricegf@antares:~/dev/202308/23-c++-templates/code_from_slides$ ./a.out
21922
ricegf@antares:~/dev/202308/23-c++-templates/code_from_slides$
```

Translating Objects to Strings

- We already have
  std::string pi = std::to_string(3.14159265);
- For any type for which operator<< is defined, we can write to_string as a template

to_string.h

```
#ifndef __TO_STRING_H
#define __TO_STRING_H

#include <ostream>
#include <sstream>

template<class T>
std::string to_string(const T& t) {
    std::ostringstream os;
    os << t;
    return os.str();
}
#endif
```

Translating Strings to Objects

- We have stoX (where X is a primitive)
  - int i = stoi(“42”); – double e = stod(“2.71828”);
- A template can again be specified to take care of input of classes which have operator>> defined

stox.h

```
#ifndef __STOX_H
#define __STOX_H

#include <istream>
#include <sstream>
#include <stdexcept>

template<class T>
T stox(const std::string& s) {
    std::istringstream is(s);
    T t;
    if (!(is >> t)) throw std::runtime_error{"stox failed for " + s};
    return t;
}
#endif
```

Class for Testing to_string and stox

test_strings.cpp

```
#include <iostream>
#include <ostream>
#include <cmath>
#include "to_string.h"
#include "stox.h"

class Coordinate {
    int _x, _y, _z;
  public:
    Coordinate(int x=0, int y=0, int z=0) : _x{x}, _y{y}, _z{z} { }
    double magnitude() {return sqrt(static_cast<double>(_x*_x+_y*_y+_z*_z));}
    friend std::ostream& operator<<(std::ostream& ost, const Coordinate& c) {
        ost << '(' << c._x << ',' << c._y << ',' << c._z << ')';
        return ost;
    }
    friend std::istream& operator>>(std::istream& ist, Coordinate& c) {
        char a;
        ist >> a; // (
        ist >> c._x;
        ist >> a; // ,
        ist >> c._y;
        ist >> a; // ,
        ist >> c._z;
        ist >> a; // )
        return ist;
    }
};
```

Continued on next page...

Translating Objects to / from Strings

test_strings.cpp

```
int main() {
    Coordinate c{3,4,5};
    std::string c_string = to_string(c);
    std::cout << "to_string: " << c_string << std::endl;

    Coordinate new_c = stox<Coordinate>("(42,7,3)");
    std::cout << "stox<Coordinate>: " << new_c << std::endl;
}
```

```
ricegf@pluto:~/dev/cpp/201808/21/strings$ make
g++ --std=c++17 -c test_strings.cpp
g++ --std=c++17 -o test_strings test_strings.o
ricegf@pluto:~/dev/cpp/201808/21/strings$ ./test_strings
to_string: (3,4,5)
stox<Coordinate>: (42,7,3)
ricegf@pluto:~/dev/cpp/201808/21/strings$
```

Another Example: *Double* Vector

- Ever wanted a vector that could store pairs of values? Me, too! It's C++ - no sooner said than written as a CSE1325 example!

dvector.h

```
template<class T, class U>
class DVector {
  public:
    void push_back(T t, U u) {tvec.push_back(t); uvec.push_back(u);}
    void pop_back() {tvec.pop_back(); uvec.pop_back();}

    T& at1(int index) {return tvec.at(index);}
    U& at2(int index) {return uvec.at(index);}

    T& back1() {return tvec.back();}
    U& back2() {return uvec.back();}

    int size() {
        assert(tvec.size() == uvec.size());
        return tvec.size();
    }
    bool empty() {return tvec.empty();}
  private:
    std::vector<T> tvec;
    std::vector<U> uvec;
};
```

Another Example: *Double* Vector

- A simple program to exercise DVector

test_dvector.h

```
#include "dvector.h"
#include <iostream>

typedef std::string Name;
typedef int Age;

int main() {
    Name name;
    Age age;
    DVector<Name, Age> ages;

    while(true) {
        std::cout << "Name? ";
        std::getline(std::cin, name);
        if(name.empty()) break;

        std::cout << "Age? ";
        std::cin >> age; std::cin.ignore();

        ages.push_back(name, age);
    }
    for(int i=ages.size()-1; i>=0; --i) {
        std::cout << ages.at1(i) << " is "
                  << ages.at2(i) << " years old" << std::endl;
    }
```

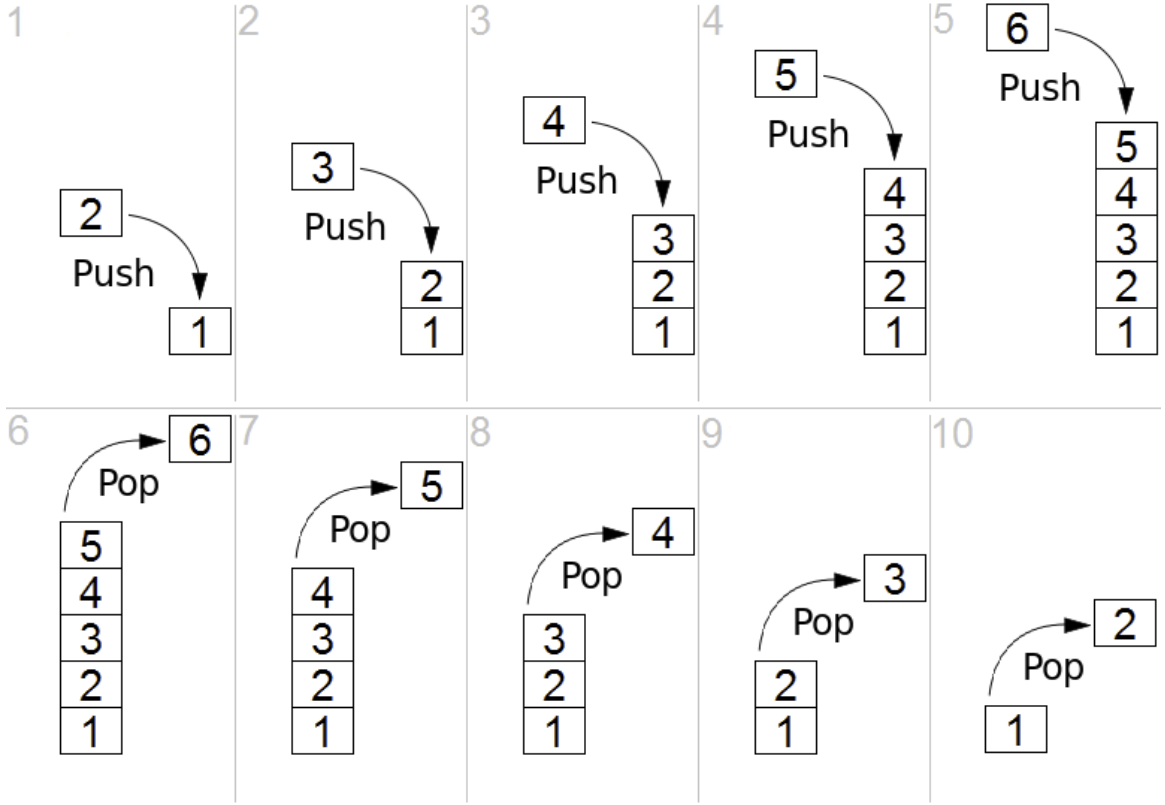
```
ricegf@pluto:~/dev/cpp/cse1325-prof/21/code_fro
rm -f *.o *.gch *~ a.out test
g++ --std=c++17 test_dvector.cpp -o test
ricegf@pluto:~/dev/cpp/cse1325-prof/21/code_fro
Name? Saanvi
Age? 21
Name? مويلع
Age? 23
Name? り く
Age? 27
Name? Alejandra
Age? 19
Name?
Alejandra is 19 years old
り く is 27 years old
مويلع is 23 years old
Saanvi is 21 years old
ricegf@pluto:~/dev/cpp/cse1325-prof/21/code_fro
```

Template Summary

- We've implemented several algorithms (max, vector, generic methods, and string conversions) independent of the types to which the algorithms should apply (generic programming) using C++ templates
- Potentially reusable algorithms that are suitable should generally be defined as templates
  - The overhead in programmer time and execution resources is very small
  - The potential reuse benefits are significant

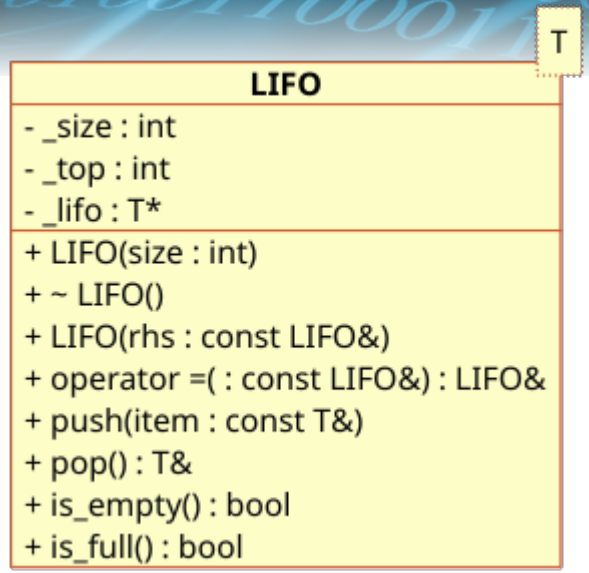
An Additional *Class* Template Example
Last-In First-Out (LIFO) Queue

- The last value pushed in is the first value popped out
- Let's implement this for ANY data type as "value"
  - int, double, vector<int>, Customer, Product*, ...



Template Class Example

- Defining our LIFO class template*
  - Specify the max number of items in the constructor, allocate an array from heap
  - Delete the heap in the destructor
  - Prevent copy constructors & assignments
  - Provide push to add a new item and pop to retrieve it
  - Provide is_empty() and is_full() Booleans to determine state of the LIFO
- Support any type of element



*In the real world, we'd build this from a container such as LinkedList

Declaration

- Situation normal except for the template line

lifo.h

Friend template function inspect will allow our demo code to show what's happening in LIFO's private variables

```
#ifndef _LIFO_H
#define _LIFO_H
#include <stdexcept>

template <class T>
class LIFO {
  public:
    LIFO(int size = 32);                      // Constructor – default to 32 items
    ~LIFO();                                  // Destructor – to clean up on deletion
    LIFO(const LIFO& rhs) = delete;           // Copy Constructor – no copies!
    LIFO& operator=(const LIFO&) = delete;    // Copy Assignment – no "=", either!
    void push(const T& item);                 // Push an item onto the LIFO
    T& pop();                                 // Pop and return an item from the LIFO
    bool is_empty() const;                    // True if LIFO has no items
    bool is_full() const;                     // True if LIFO cannot hold another item
    template <class U>
    friend void inspect(LIFO<U>& lifo);       // Hook for testing
private:
    int _size;                                // Number of elements in the LIFO array
    int _top;                                 // Index of last item pushed to LIFO
    T* _lifo;                                 // The actual LIFO array
};
// continued next slide
```

Implementation

- Each definition includes a template declaration

lifo.h

```
// continued from last slide

template <class T>
LIFO<T>::LIFO(int size) : _size{size}, _top{-1}, _lifo{new T[size]} { }

template <class T>
LIFO<T>::~LIFO() { delete[] _lifo; }

template <class T>
void LIFO<T>::push(const T& item) {
  if (!this->is_full()) _lifo[++_top] = item;
  else throw std::runtime_error("LIFO stack overflow");
}
template <class T>
T& LIFO<T>::pop() {
  if (!this->is_empty()) return _lifo[_top--];
  else throw std::runtime_error("LIFO stack underflow");
}
template <class T>
bool LIFO<T>::is_empty() const { return (_top == -1); }

template <class T>
bool LIFO<T>::is_full() const { return (_top == _size-1); }

#endif
```

Declaration ...in the *Header File???*

lifo.h

```
// same as previous slide

template <class T>
LIFO<T>::LIFO(int size) : _size{size}, _top{-1}, _lifo{new T[size]} { }

template <class T>
LIFO<T>::~LIFO() { delete[] _lifo; }

template <class T>
void LIFO<T>::push(const T& item) {
  if (!this->is_full()) _lifo[++_top] = item;
  else throw std::runtime_error("LIFO stack overflow");
}
template <class T>
T& LIFO<T>::pop() {
  if (!this->is_empty()) return _lifo[_top--];
  else throw std::runtime_error("LIFO stack underflow");
}
template <class T>
bool LIFO<T>::is_empty() const { return (_top == -1); }

template <class T>
bool LIFO<T>::is_full() const { return (_top == _size-1); }

#endif
```

Templates Live in the .h

- The rule has been “Declare in .h, Define in .cpp”
  - But templates are a different animal – they are all declarations *until they are instanced*
  - Once instanced, C++ begins actually generating code (using the parameterized type)
  - Thus you can safely put the entire definition of the template in the .h with no duplicate definition errors
- And you must!
  - Otherwise, C++ will not have the *definition* of the template from which to generate code when you instance the template! You get linker errors...

Templates Live in the .h

- Splitting a template between .h and .cpp results in linker errors like this:

```
test_lifo.o: In function `test_underflow()':
test_lifo.cpp:(.text+0x49c): undefined reference to `LIFO<double>::LIFO(int)'
test_lifo.cpp:(.text+0x4bc): undefined reference to `LIFO<double>::push(double const&)'
test_lifo.cpp:(.text+0x4c8): undefined reference to `LIFO<double>::pop()'
test_lifo.cpp:(.text+0x4d4): undefined reference to `LIFO<double>::pop()'
test_lifo.cpp:(.text+0x4fc): undefined reference to `LIFO<double>::~LIFO()'
test_lifo.cpp:(.text+0x52c): undefined reference to `LIFO<double>::~LIFO()'
```

Bottom Line

- For *normal code*, put declarations in .h and definitions in .cpp
- For *templates*, put declarations AND definitions in .h

Testing our LIFO Template

```
#include "lifo.h"
#include <iostream>

void test_strings() {  // Basic test – push some text in, pop it back out
  LIFO<std::string> lifo;
  lifo.push("Hello");
  lifo.push("Goodbye");
  if (lifo.pop() != "Goodbye")
    std::cerr << "FAIL: std::string 'Goodbye' did not pop" << std::endl;
  if (lifo.pop() != "Hello")
    std::cerr << "FAIL: std::string 'Hello' did not pop" << std::endl;
}

void test_bool_methods() { // Verify is_full / is_empty + non-default constructor
  LIFO<std::string> lifo{3};
  if (!lifo.is_empty())
    std::cerr << "FAIL: New bool LIFO was not empty" << std::endl;
  if (lifo.is_full())
    std::cerr << "FAIL: New bool LIFO was full" << std::endl;
  lifo.push("Larry");
  lifo.push("Curly");
  lifo.push("Moe");
  if (lifo.is_empty())
    std::cerr << "FAIL: Full bool LIFO was empty" << std::endl;
  if (!lifo.is_full())
    std::cerr << "FAIL: Full bool LIFO was not full" << std::endl;
}

// continued on next slide
```

Testing our LIFO Template

```
// continued from last slide

void test_underflow() { // Verify runtime exception on underflow
  LIFO<double> lifo;
  lifo.push(1);
  lifo.pop();
  try {
    lifo.pop();
    std::cerr << "FAIL: Underflow did not produce exception" << std::endl;
  } catch(...) {
  }
}

void test_overflow() {  // Verify runtime exception on overflow
  LIFO<int> lifo{3};

  lifo.push(1);
  lifo.push(2);
  lifo.push(3);
  try {
    lifo.push(4);
    std::cerr << "FAIL: Overflow did not produce exception" << std::endl;
  } catch(...) {
  }
}
// continued on next slide
```

Testing our LIFO Template

```
// continued from last slide

// Since our deleted copy constructor and copy assignment operator tests
// cause compile errors if “successful”, we wrap them in preprocessor conditionals.
// See the Makefile for how to set this as part of the build!

#ifdef _TEST_DELETES_
void copy_constructor(LIFO<int> x) {
  std::cout << x.pop() << std::endl;
}

void test_copy_constructor_and_assignment() {  // Verify no copies
  LIFO<int> lifo1;
  lifo1.push(1);
  copy_constructor(lifo1);
  LIFO<int> lifo2;
  lifo2 = lifo1;
}
#endif

int main() {  // Run the regression tests!
  test_strings();
  test_bool_methods();
  test_underflow();
  test_overflow();
}
```

Our Makefile

```
# Use ‘make’ to create a regression test binary
# Use ‘make delete’ to verify that any copy attempts cause a compile error

CXXFLAGS += --std=c++17

all: test_lifo demo_lifo

debug: CXXFLAGS += -g
debug: test_lifo

delete: CXXFLAGS += -D_TEST_DELETES_  # This defines  preprocessor var _TEST_DELETES_
delete: rebuild

rebuild: clean all

test_lifo: test_lifo.o
    g++ $(CXXFLAGS) -o test_lifo test_lifo.o
test_lifo.o: test_lifo.cpp lifo.h
    g++ $(CXXFLAGS) -c test_lifo.cpp
demo_lifo: demo_lifo.o
    g++ $(CXXFLAGS) -o demo_lifo demo_lifo.o
demo_lifo.o: demo_lifo.cpp lifo.h
    g++ $(CXXFLAGS) -c demo_lifo.cpp
clean:
    -rm -f *.o *.gch *~ test_lifo demo_lifo
```

Test Results (SUCCESS!)

```
ricegf@pluto:~/dev/cpp/201808/21/LIFO$ make rebuild
rm -f *.o *.gch *~ test_lifo demo_lifo
g++ --std=c++17 -c test_lifo.cpp
g++ --std=c++17 -o test_lifo test_lifo.o
g++ --std=c++17 -c demo_lifo.cpp
g++ --std=c++17 -o demo_lifo demo_lifo.o
ricegf@pluto:~/dev/cpp/201808/21/LIFO$ ./test_lifo
ricegf@pluto:~/dev/cpp/201808/21/LIFO$ make delete
rm -f *.o *.gch *~ test_lifo demo_lifo
g++ --std=c++17 -D_TEST_DELETES_   -c test_lifo.cpp
test_lifo.cpp: In function 'void test_copy_constructor_and_assignment()':
test_lifo.cpp:65:25: error: use of deleted function 'LIFO<T>::LIFO(const LIFO<T>&) [with T = i
nt]'
   copy_constructor(lifo1);
                    ^
In file included from test_lifo.cpp:1:0:
lifo.h:10:5: note: declared here
     LIFO(const LIFO& rhs) = delete;         // Copy Constructor – no copies!
     ^
test_lifo.cpp:58:6: note:   initializing argument 1 of 'void copy_constructor(LIFO<int>)'
 void copy_constructor(LIFO<int> x) {
      ^
test_lifo.cpp:67:9: error: use of deleted function 'LIFO<T>& LIFO<T>::operator=(const LIFO<T>&
) [with T = int]'
   lifo2 = lifo1;
         ^
In file included from test_lifo.cpp:1:0:
lifo.h:11:11: note: declared here
     LIFO& operator=(const LIFO&) = delete;  // Copy Assignment – no "=", either!
           ^
Makefile:19: recipe for target 'test_lifo.o' failed
make: *** [test_lifo.o] Error 1
ricegf@pluto:~/dev/cpp/201808/21/LIFO$
```

The "normal" regression tests prints no FAILs, and so passes.

We expect these errors if code tries to copy our FILO. "make delete" enables compilation of code that does just that, so this test passes, too.

Demonstrating our LIFO Template
Inspector and a Customer Class

```
#include "lifo.h"
#include <iostream>
#include <ostream>
#include <cmath>

class Coordinate {
    int _x, _y, _z;
  public:
    Coordinate(int x=0, int y=0, int z=0) : _x{x}, _y{y}, _z{z} { }
    double magnitude() {return sqrt(static_cast<double>(_x*_x+_y*_y+_z*_z));}
    friend std::ostream& operator<<(std::ostream& ost, const Coordinate& c) {
        ost << '(' << c._x << ',' << c._y << ',' << c._z << ')';
        return ost;
    }
};

template<class T>
void inspect(LIFO<T>& lifo) {
    std::cout << "LIFO: ";
    for (int i=0; i<=lifo._top; ++i) std::cout << lifo._lifo[i] << ' ';
    std::cout << std::endl;
}
```

Our custom 3D Coordinate class, to prove our LIFO template works just fine with our own custom classes, too!

We provide a definition for our inspect template (a friend of LIFO's) that prints out the current contents of the LIFO stack, whatever types they may be.

Demonstrating our LIFO Template

An Integer LIFO

```
// continued from previous slide

int main() {
    std::cout << "Using int" << std::endl << "=========" << std::endl;
    {
        LIFO<int> lifo;
        inspect(lifo);
        lifo.push(3); std::cout << "Push 3, "; inspect(lifo);
        lifo.push(1); std::cout << "Push 1, "; inspect(lifo);
        lifo.push(4); std::cout << "Push 4, "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        lifo.push(1); std::cout << "Push 1, "; inspect(lifo);
        lifo.push(5); std::cout << "Push 5, "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
    }
    std::cout << std::endl;

// continued on next slide
```

```
Using int
=========
LIFO:
Push 3, LIFO: 3
Push 1, LIFO: 3 1
Push 4, LIFO: 3 1 4
Pop 4, LIFO: 3 1
Pop 1, LIFO: 3
Push 1, LIFO: 3 1
Push 5, LIFO: 3 1 5
Pop 5, LIFO: 3 1
Pop 1, LIFO: 3
Pop 3, LIFO:
```

Demonstrating our LIFO Template

A String LIFO

```
// continued from previous slide

    std::cout << "Using string" << std::endl << "============" << std::endl;
    {
        LIFO<std::string> lifo;
        inspect(lifo);
        lifo.push("Now"); std::cout << "Push Now, "; inspect(lifo);
        lifo.push("is"); std::cout << "Push is, "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        lifo.push("the"); std::cout << "Push the, "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        lifo.push("time"); std::cout << "Push time, "; inspect(lifo);
        lifo.push("for"); std::cout << "Push for, "; inspect(lifo);
        lifo.push("all"); std::cout << "Push all, "; inspect(
        std::cout << "Pop " << lifo.pop() << ", "; inspect(
        std::cout << "Pop " << lifo.pop() << ", "; inspect(
        std::cout << "Pop " << lifo.pop() << ", "; inspect(
        std::cout << "Pop " << lifo.pop() << ", "; inspect(
    }
    std::cout << std::endl;

// continued on next slide
```

```
Using string
============
LIFO:
Push Now, LIFO: Now
Push is, LIFO: Now is
Pop is, LIFO: Now
Push the, LIFO: Now the
Pop the, LIFO: Now
Push time, LIFO: Now time
Push for, LIFO: Now time for
Push all, LIFO: Now time for all
Pop all, LIFO: Now time for
Pop for, LIFO: Now time
Pop time, LIFO: Now
Pop Now, LIFO:
```

Demonstrating our LIFO Template

A Custom Coordinate Class LIFO

```
// continued from previous slide

    std::cout << "Using custom class Coordinate" << std::endl <<
"============================" << std::endl;
    {
        LIFO<Coordinate> lifo;
        inspect(lifo);
        lifo.push(Coordinate(1,5,2)); std::cout << "Push (1,5,2), "; inspect(lifo);
        lifo.push(Coordinate(3,18,7)); std::cout << "Push (3,18,7), "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        lifo.push(Coordinate(42,7,3)); std::cout << "Push (42,7,3), "; inspect(lifo);
        lifo.push(Coordinate(3,7,42)); std::cout << "Push (3,7,42), "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
        std::cout << "Pop " << lifo.pop() << ", "; inspect(lifo);
    }
}
```

```
Using custom class Coordinate
============================
LIFO:
Push (1,5,2), LIFO: (1,5,2)
Push (3,18,7), LIFO: (1,5,2) (3,18,7)
Pop (3,18,7), LIFO: (1,5,2)
Push (42,7,3), LIFO: (1,5,2) (42,7,3)
Push (3,7,42), LIFO: (1,5,2) (42,7,3) (3,7,42)
Pop (3,7,42), LIFO: (1,5,2) (42,7,3)
Pop (42,7,3), LIFO: (1,5,2)
Pop (1,5,2), LIFO:
ricegf@pluto:~/dev/cpp/201808/21/LIFO$
```

Other Template Uses

- Where else have we written the same code over... and over... and over... again?