CSE 1325: Object-Oriented Programming

Lecture 20 / Chapters 17-19

# Templates


Mr. George F. Rice

george.rice@uta.edu


Based on material by Bjarne Stroustrup

www.stroustrup.com/Programming


ERB 402

Office Hours:

Tuesday Thursday 11 - 12

Or by appointment

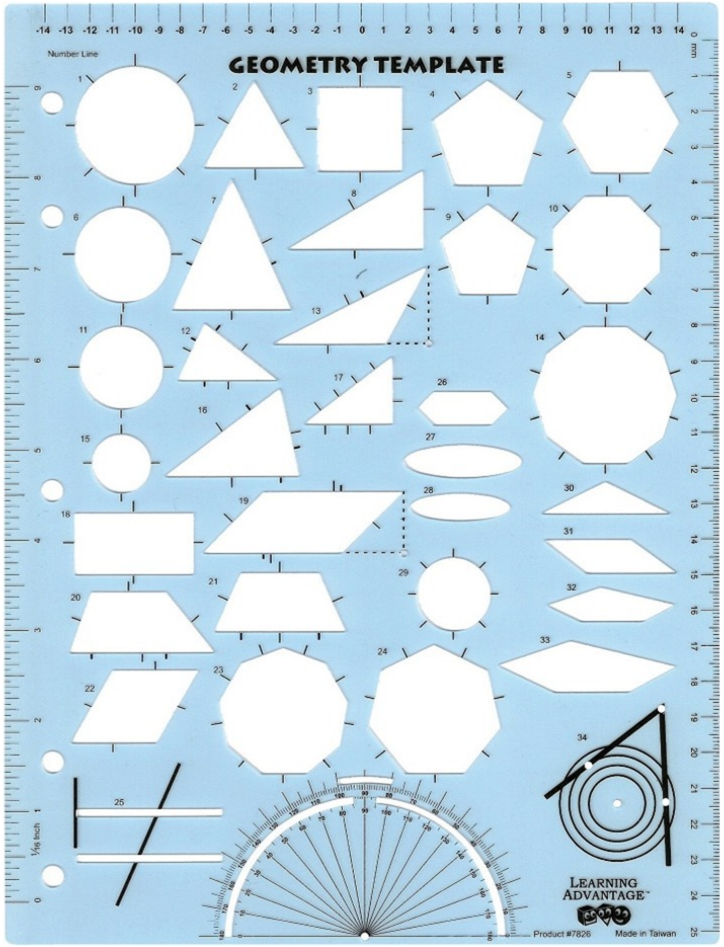
Lecture 19

# Quick Review

- The provision of a single interface to multiple derived classes, enabling the same method call to invoke different derived methods to generate different results is called **polymorphism**.
- In C++, this requires the **virtual** keyword for base class methods, and also the use of **pointers** or **references** for base class variables storing derived objects.
- The closest feature that C++ has to providing raw memory is the **void***.
- Changing the type of an object is called a **cast**. Describe 3 types, and when each should be used in a C++ program. (1) C-style cast, virtually never (2) static_cast, for non-polymorphic sources, and (3) dynamic_cast, for polymorphic sources
- Define the difference between stack and heap memory, and how each is allocated. Stack is allocated for non-class scoped variables, allocated by a definition and deleted when exiting scope. Heap is allocated via “new” and deleted via “delete” or “delete[ ]”
  - If new cannot allocate memory (for example, because there is not enough memory available), what does C++ do? Throw a bad_alloc exception
- Give 2 examples of errors that may cause a memory leak. (1) Pointer on stack goes out of scope (2) Pointer value is overwritten by another “new”
- Identify 3 strategies to avoid memory leaks. (1) Allocate memory in the constructor, de-allocate it in the corresponding destructor (2) Use smart pointers (3) Use Gtk::manage
- **delete[ ]** de-allocates arrays from the heap, while **delete** de-allocates normal variables.

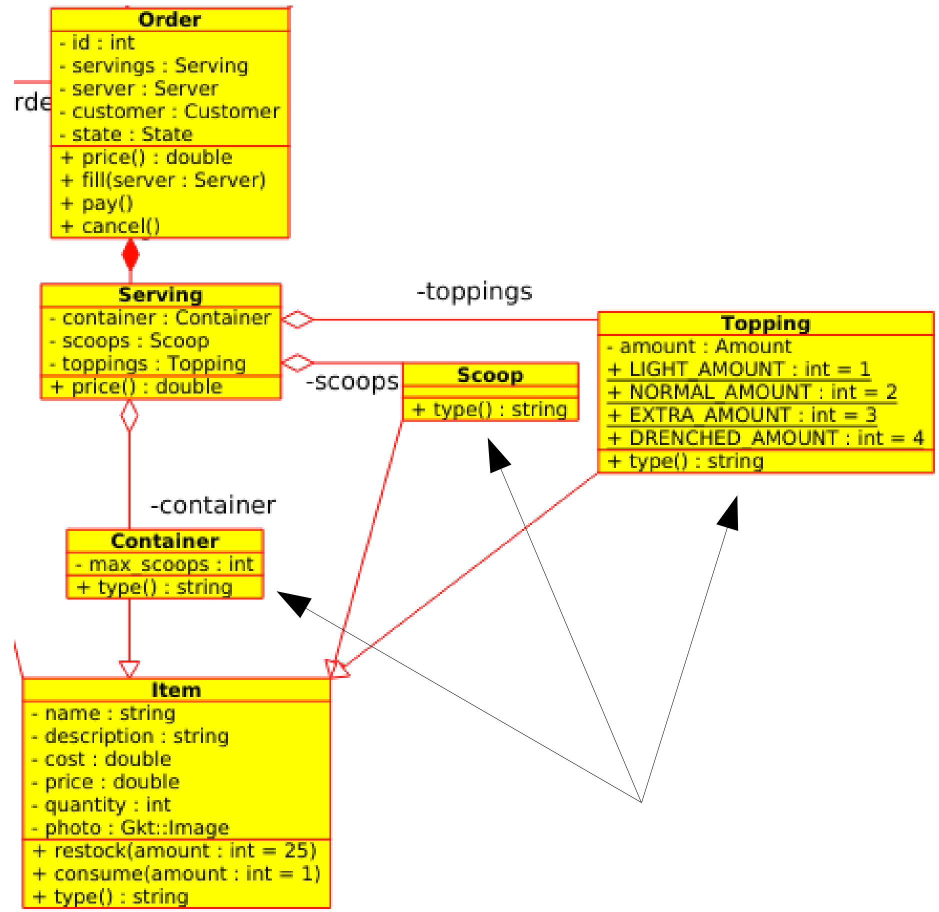
# Overview: Templates and Iterators

- Templates
  - Alternates
  - C++ Implementation
  - Example
- Standard Template Library (STL)
- Iterators



# A Challenge

- The suggested solution includes Container, Scoop, and Topping classes derived from Item
- CHALLENGE: Write command line interfaces for the user to select from a list of each



# Dealing with Similarities

- Write a function that returns an index to a Container, given a vector of Container pointers and a get_int function

```
int get_container(vector<Container*> containers) {
  cout << "(-1) None" << endl;
  for (int i= 0; i < containers.size(); ++i) {
    cout << "(" << i << ") " << containers[i] << endl;
  }
  return get_int("Select a : ", -1, containers.size()-1); // get int from user in range
}
```

- Now we need them for the other 2 item types...
  - The *algorithm* is the same, but the *types* differ
  - Polymorphism doesn't help, because a vector<Item*> parameter can't accept a vector<Container*> object

# (Non)-Option #1: Dynamic Languages

- Dynamic languages (such as Python) do not enforce types
  - "Duck typing" - any variable may hold any object type, provided it offers the needed methods
  - Downside: No clear type specifications
  - Downside: If any object DOESN'T offer the needed methods, you find out a *runtime*, not compile time

```
def get_item(item, items):  # string, list of Item et. al. objects
  print("(-1) None")
  for i in range(0, len(items)):
    print("({0}) {1}".format(i, items[i]))
  return(get_int("Select a {1}: ".format(item), -1, len(items)))
```

But we don't have a dynamic language. Moving on...

# Option #2: Duplicate and Edit the Code

- Sometimes brute force is the best answer
  - But not this time – since bugs are proportional to the number of lines of code, we're duplicating <u>bugs</u>

```
int get_container(vector<Container*> items) {
  cout << "(-1) None" << endl;
  for (int i= 0; i < items.size(); ++i) {
    cout << "(" << i << ") " << items[i] << endl;
  }
  return get_int("Select a Container: ", -1, items.size()-1);
}

int get_container(vector<Topping*> items) {
  cout << "(-1) None" << endl;
  for (int i= 0; i < items.size(); ++i) {
    cout << "(" << i << ") " << items[i] << endl;
  }
  return get_int("Select a Topping: ", -1, items.size()-1);
}
```

We don't want more bugs. Moving on...

# Option #3: Write a Code Generator

- We can automate the code duplication
  - The bugs are now solo in the code generator
  - Downside – This complicates the Makefile and makes learning to support the code more difficult

```
#!/usr/bin/env python
def gen_getter(type): # string
  print(R"""int get_item(vector<{0}*> items) {
  cout << "(-1) None" << endl;
  for (int i= 0; i < items.size(); ++i) {
    cout << "(" << i << ") " << items[i] << endl;
  }
  return get_int("Select a {0}: ", -1, items.size()-1);
}""".format(type)

gen_getter("Container")
gen_getter("Flavor")
gen_getter("Topping")
```

We don’t want complicated support. Moving on...

# Option #4: Use the Preprocessor

- The preprocessor lexically mangles the code before passing it to the compiler
  - This is essentially a built-in code generator
  - Downside – The preprocessor is tricky and non-obvious, particularly in the *debugger*

```
#define getter(item) \
int get_item(vector<##item*> items) { \
  cout << "(-1) None" << endl; \
  for (int i= 0; i < items.size(); ++i) { \
    cout << "(" << i << ") " << items[i] << endl; \
  } \
  return get_int("Select a ##item: ", -1, items.size()-1); \
} \

getter(Container)
getter(Flavor)
getter(Topping)
```

We don’t want tricky and non-obvious. Moving on...

# Option #5: Use 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

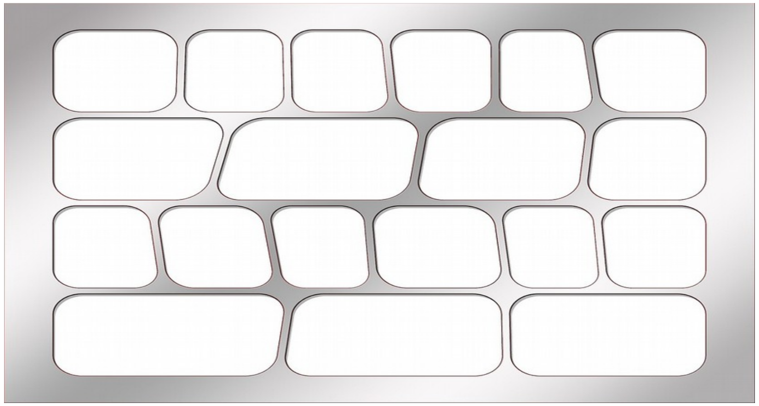
```
template <class T>
int get_item(Items item, vector<T*> items) {
  cout << "(-1) None" << endl;
  for (int i= 0; i < items.size(); ++i) {
    cout << "(" << i << ") " << items[i] << endl;
  }
  return get_int("Select a " + item_name(item) + ": ", -1, items.size()-1);
}

int selection = get_item<Container>(Items::CONTAINER, container_vector);
int selection = get_items<Flavor>(Items::FLAVOR, flavor_vector);
int selection = get_items<Topping>(Items::TOPPING, topping_vector);
```

This keeps only one copy of the bugs
and clearly signals that template substitution is taking place

# Templates

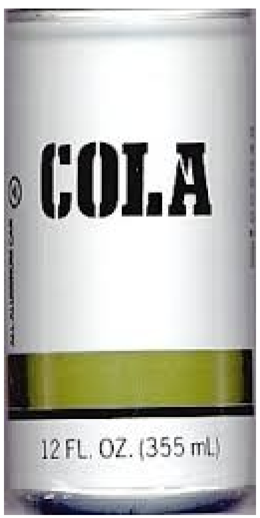
- **Template** – A C++ construct representing a function or class in terms of generic types


  - This enables the algorithm to be written independent of the types of data to which it applies
  - A type must be specified when the template class is instanced or the template function is called
- As with dynamic languages, the type specified must supply the needed methods
  - For example, a templated sort algorithm might only work if the type supplied defines operator<



# Generic Programming

- **Generic Programming** – Writing algorithms in terms of types that are specified as parameters during instantiation or invocation

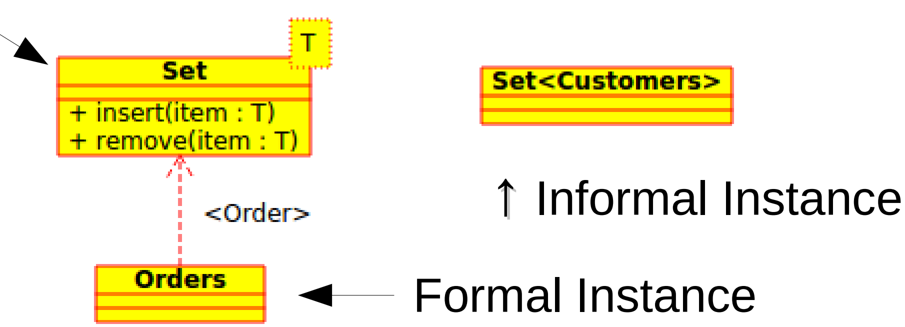




  - In C++, generic programming may apply to functions or classes
  - We’re obviously more interested in the classes...

# Other Names for Templates

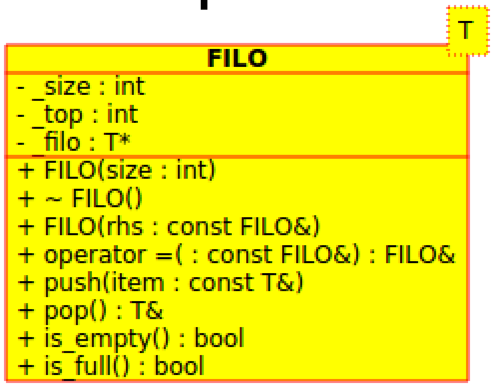
- Generics
  - Ada (which first supported the technique), C#, Java, and Objective-C
- Parametric Polymorphism
  - Scala, Haskell
  - Also sometimes called **Compile-Time Polymorphism**
- Parameterized Types
  - Design patterns

**UML Notation**



# Template Class Example

- Let’s define a FILO (also called LIFO) class template*
  - Specify the max number of items in the constructor and allocate an array from heap
  - Delete the heap in the destructor
  - Prevent copy constructors and 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 FILO
- Support any type of element




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

# Declare the Interface in .h

- Situation normal except for the template line

```
#ifndef _FILO_H
#define _FILO_H

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

# Implement the Interface in .h

- Wait... What??? In the *header file*?

```
// continued from last slide
#include <stdexcept>
using namespace std;

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

template <class T>
FILO<T>::~FILO() { delete[] _filo; }

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

```
ricegf@pluto:~/dev/cpp/201701/23$ g++ -std=c++11 -c test_filo.cpp
ricegf@pluto:~/dev/cpp/201701/23$ g++ -std=c++11 -c filo.cpp
ricegf@pluto:~/dev/cpp/201701/23$ g++ -std=c++11 test_filo.o filo.o
test_filo.o: In function `main':
test_filo.cpp:(.text+0x16): undefined reference to `FILO<int>::FILO(int)'
test_filo.cpp:(.text+0x30): undefined reference to `FILO<int>::push(int const&)'
test_filo.cpp:(.text+0x4a): undefined reference to `FILO<int>::push(int const&)'
test_filo.cpp:(.text+0x64): undefined reference to `FILO<int>::push(int const&)'
test_filo.cpp:(.text+0x70): undefined reference to `FILO<int>::pop()'
test_filo.cpp:(.text+0x97): undefined reference to `FILO<int>::pop()'
test_filo.cpp:(.text+0xbe): undefined reference to `FILO<int>::pop()'
test_filo.cpp:(.text+0xe5): undefined reference to `FILO<int>::~FILO()'
test_filo.cpp:(.text+0xfb): undefined reference to `FILO<int>::~FILO()'
collect2: error: ld returned 1 exit status
```

Bottom Line

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

# Testing our FILO Template

```
#include "filo.h"
#include <iostream>
using namespace std;

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

void test_bool_methods() { // Verify is_full / is_empty + non-default constructor
  FILO<string> filo{3};
  if (!filo.is_empty())
    cerr << "FAIL: New bool FILO was not empty" << endl;
  if (filo.is_full())
    cerr << "FAIL: New bool FILO was full" << endl;
  filo.push("Larry");
  filo.push("Curly");
  filo.push("Moe");
  if (filo.is_empty())
    cerr << "FAIL: Full bool FILO was empty" << endl;
  if (!filo.is_full())
    cerr << "FAIL: Full bool FILO was not full" << endl;
}
// continued on next slide
```

# Testing our FILO Template

```
// continued from last slide

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

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

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

# Testing our FILO 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(FILO<int> x) {
  cout << x.pop() << endl;
}

void test_copy_constructor_and_assignment() {  // Verify no copies
  FILO<int> filo1;
  filo1.push(1);
  copy_constructor(filo1);
  FILO<int> filo2;
  filo2 = filo1;
}
#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++14

all: main

debug: CXXFLAGS += -g
debug: main

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

rebuild: clean main

main: test_filo.o
        g++ $(CXXFLAGS) test_filo.o
test_filo.o: test_filo.cpp filo.h
        g++ $(CXXFLAGS) -c test_filo.cpp
clean:
        -rm -f *.o *~
```

# Test Results (SUCCESS!)

```
ricegf@pluto:~/dev/cpp/201701/23/filo$ make clean
rm -f *.o *~
ricegf@pluto:~/dev/cpp/201701/23/filo$ make delete
g++ -std=c++11  -D_TEST_DELETES_ -c test_filo.cpp
test_filo.cpp: In function ‘void test_copy_constructor_and_assignment()’:
test_filo.cpp:62:25: error: use of deleted function ‘FILO<T>::FILO(const FILO<T>&) [with T = int]’
   copy_constructor(filo1);
                         ^
In file included from test_filo.cpp:1:0:
filo.h:8:5: error: declared here
     FILO(const FILO& rhs) = delete;
     ^
test_filo.cpp:55:6: error:   initializing argument 1 of ‘void copy_constructor(FILO<int>)’
 void copy_constructor(FILO<int> x) {
      ^
test_filo.cpp:64:9: error: use of deleted function ‘FILO<T>& FILO<T>::operator=(const FILO<T>&) [with
T = int]’
   filo2 = filo1;
         ^
In file included from test_filo.cpp:1:0:
filo.h:9:11: error: declared here
     FILO& operator=(const FILO&) = delete;
           ^
make: *** [test_filo.o] Error 1
ricegf@pluto:~/dev/cpp/201701/23/filo$ make clean
rm -f *.o *~
ricegf@pluto:~/dev/cpp/201701/23/filo$ make
g++ -std=c++11  -c test_filo.cpp
g++ -std=c++11  test_filo.o
ricegf@pluto:~/dev/cpp/201701/23/filo$ ./a.out
ricegf@pluto:~/dev/cpp/201701/23/filo$
```

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.

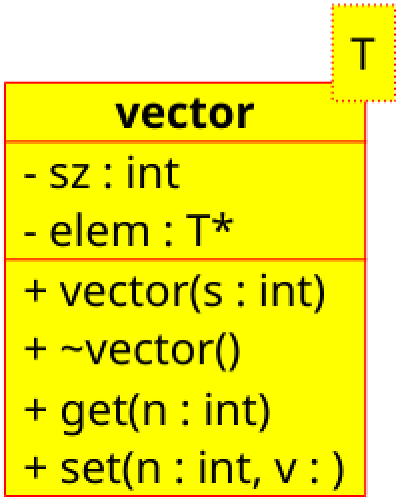
The “normal” regression tests prints no FAILs, and so also passes

# Adding Template to Vector v0.3

```
// a very simplified vector of doubles:
class vector {
   int sz;       // the size
   double* elem; // a pointer to the elements
public:
   vector(int s) : sz(s), elem{new double[s]} { }
   ~vector() {delete[ ] elem; }
   double get(int n) const { return elem[n]; }  // access: read
   void set(int n, double v) { elem[n]=v; }     // access: write
};
```

- How would we recast our simplistic “vector of doubles” into a more useful “vector template”?
  - **First**, design the UML class specification
  - **Next**, implement the code
  - **Last**, instance and test

# UML Class Diagram for Vector



# Vector v0.4 – Now a Template

```
template <class T>
class vector {
   int sz;       // the size
   T* elem;      // a pointer to the elements
public:
   vector(int s) : sz(s), elem{new T[s]} { }
   ~vector() {delete[ ] elem; }
   T get(int n) const { return elem[n]; }  // access: read
   void set(int n, T v) { elem[n]=v; }     // access: write
};
```

- Add the template specifier before the class declaration
- Replace double with T
- That's it!

# Testing Vector v0.4 - ints

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

int main() {
    vector<int> vi{10};
    for (int i=0; i<10; ++i)
        vi.set(i, i*i);

    for (int i=9; i>=0; --i) {
        if (vi.get(i) != i*i) {
            std::cerr << "Fail: stored " << vi.get(i)
                      << " instead of " << i*i << std::endl;
        }
    }
}
```

```
ricegf@pluto:~/dev/cpp/201801/20$ g++ --std=c++14 test_vector.cpp
ricegf@pluto:~/dev/cpp/201801/20$ ./a.out
ricegf@pluto:~/dev/cpp/201801/20$ _
```

# Testing Vector v0.4 - Scoops

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

int main() {
    // test with int omitted
    vector<Scoop> vs{4};
    vs.set(2,
        Scoop{"Chocolate", "Brown cocoa goodness", 0.25, 0.75}
          );

    if (  vs.get(2).name()        != "Chocolate"
       || vs.get(2).description() !="Brown cocoa goodness"
       || vs.get(2).cost()        != 0.25
       || vs.get(2).price()       != 0.75) {
         std::cerr << "Fail: Scoop" << std::endl;
    }
}
```

```
ricegf@pluto:~/dev/cpp/201801/20$ g++ --std=c++14 *.cpp
In file included from test_vector.cpp:1:0:
vector.h: In instantiation of 'vector<T>::vector(int) [with T = Scoop]':
test_vector.cpp:17:23:   required from here
vector.h:6:40: error: no matching function for call to 'Scoop::Scoop()'
    vector(int s) : sz(s), elem{new T[s]} { }
                                        ^
In file included from test_vector.cpp:2:0:
scoop.h:5:5: note: candidate: Scoop::Scoop(std::__cxx11::string, std::__cxx11::string, double,
 double)
     Scoop(std::string name, std::string description, double cost, double price);
     ^
scoop.h:5:5: note:   candidate expects 4 arguments, 0 provided
```

**What's the error?**
**Options to fix?**

# Identifying and Fixing a Bug

- Scoop has no default constructor
  - The vector constructor creates an array of Scoops, which requires a default constructor

```
vector(int s) : sz(s), elem{new T[s]} { }
```

- Among options to fix
  - Limit vector to only types with a default constructor
  - Create an array of *pointers* to Scoops, which can be filled in via the vector::set method
- We'll add a default constructor for Scoop

```
Scoop::Scoop() : Item("", "", 0.0, 0.0) { }
```

# Testing Vector v0.4 – Scoops v2

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

int main() {
    // test with int omitted
    vector<Scoop> vs{4};
    vs.set(2,
        Scoop{"Chocolate", "Brown cocoa goodness", 0.25, 0.75}
          );

    if (  vs.get(2).name()        != "Chocolate"
       || vs.get(2).description() !="Brown cocoa goodness"
       || vs.get(2).cost()        != 0.25
       || vs.get(2).price()       != 0.75) {
         std::cerr << "Fail: Scoop" << std::endl;
    }
}
```

```
ricegf@pluto:~/dev/cpp/201801/20$ vi scoop.cpp
ricegf@pluto:~/dev/cpp/201801/20$ vi scoop.h
ricegf@pluto:~/dev/cpp/201801/20$ !g++
g++ --std=c++14 *.cpp
ricegf@pluto:~/dev/cpp/201801/20$ ./a.out
ricegf@pluto:~/dev/cpp/201801/20$
```

# Template Summary

- We've implemented an algorithm (First In Last Out) independent of the type of items to which the algorithm 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

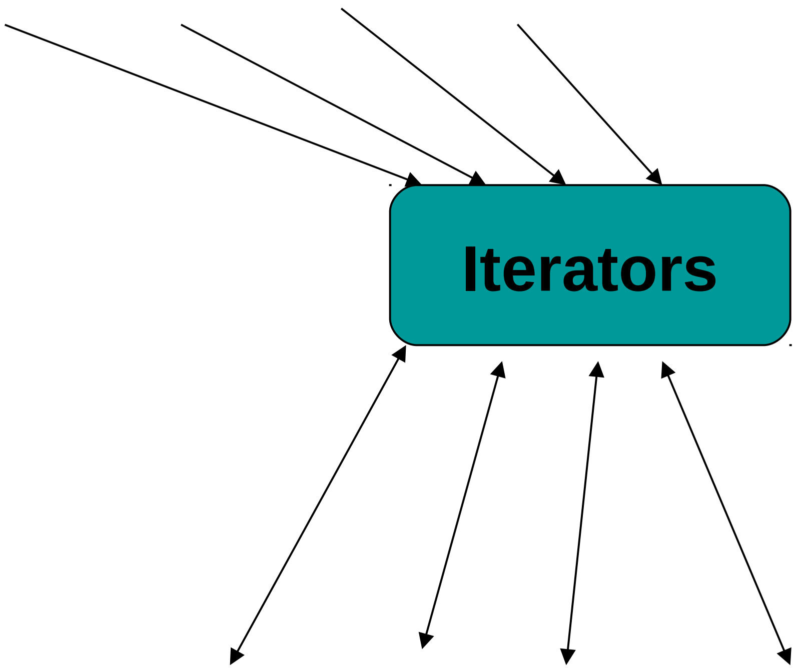
# The Standard Template Library (STL)

- The STL provides a good variety of well-implemented, mostly non-numerical algorithms focused on organizing code and data as C++ templates
  - Performance is a key goal
  - Our friend vector is an STL member
- Originally designed by Alex Stepanov
  - His goal was to make good programming “like math”

# Basic STL Model

## Algorithms

Sort, find, search, copy, ...




vector, list, map, unordered_map, ...

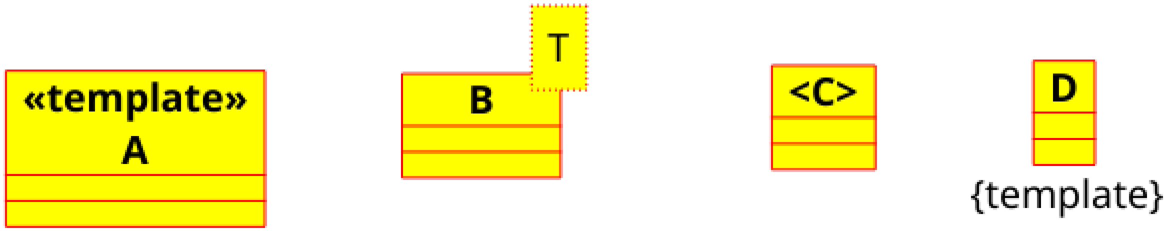
## Containers

- Separation of concerns
  - Algorithms manipulate data, but don't know about containers
  - Containers store data, but don't know about algorithms
  - Algorithms and containers interact through iterators
    - Each container has its own iterator types

The STL also defines some **Functors**, but we'll ignore those...

# Quick Review

- Writing algorithms in terms of types that are specified as parameters during instantiation or invocation is called __________ ______________.
- A ______ is a C++ construct representing a function or class in terms of generic types.
- True or False: Both the declaration AND definition of a template class must be placed in the header file.
- The ________ _________ __________ provides a good variety of well-implemented, mostly non-numerical algorithms focused on organizing code and data as C++ templates.
- A template is represented in the UML as:



# For Next Class

- (Optional) Read Chapters 17-19 in Stroustrop
  - Do the Drills!
- Skim Chapters 20-21 for next class
- We’ll explore the template containers provided by the STL, and learn (much) more about iterators
- **Sprint #2 now in progress: GUI Time!**