

C++ Templates and the STL

CSE 333

Instructor: Alex Sanchez-Stern

Teaching Assistants:

Audrey Seo

Deeksha Vatswani

Derek de Leuw

Katie Gilchrist

Administrivia

- ❖ Homework 2 due **tomorrow night** (7/18)
 - File system crawler, indexer, and search engine
 - **Don't forget to clone your repo to double-/triple-/quadruple-check compilation, execution, and tests!**
 - If your code won't build or run when we clone it, well ... you should have caught that ...
- ❖ Midterm is on Monday
 - Extra points for coming to office hours; see last Wednesdays slides for details

Lecture Outline

- ❖ **Templates**
- ❖ The C++ Standard Template Library

Suppose that...

- ❖ You want to write a function to compare two ints
- ❖ You want to write a function to compare two doubles
- Function overloading!

```
// 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 double &value1, const double &value2) {  
    if (value1 < value2) return -1;  
    if (value2 < value1) return 1;  
    return 0;  
}
```

Hm...

- ❖ The two implementations of **compare** are nearly identical!
 - What if we wanted a version of **compare** for *every* comparable type?
 - We could write (many) more functions, but that's obviously wasteful and redundant

- ❖ What we'd prefer to do is write “*generic code*”
 - Code that is **type-independent**

C++ Parametric Polymorphism

- ❖ C++ has the notion of **templates**
 - A function or class that accepts a ***type*** as a parameter
 - You define the function or class once in a type-agnostic way
 - When you invoke the function or instantiate the class, you specify (one or more) types as arguments to it

Function Templates

- ❖ Template to **compare** two “things”:

Syntax to start a template; T can be any variable name.

Use T instead of the type

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

functiontemplate.cc

Function Templates

- ❖ Template to **compare** two “things”:

```
#include <iostream>
#include <string>

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

int main(int argc, char **argv) {
    std::cout << compare<int>(10, 20) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    std::cout << compare<std::string>("Hello", "World") << std::endl;
    return EXIT_SUCCESS;
}
```

functiontemplate.cc

```
> ./functiontemplate
-1
-1
-1
```

Compiler Inference

- ❖ Similar, but letting the compiler infer the types:

```
#include <iostream>
#include <string>

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

int main(int argc, char **argv) {
    std::cout << compare(10, 20) << std::endl; // ok
    std::cout << compare(50.5, 50.6) << std::endl; // ok
    std::cout << compare("Hello", "World") << std::endl; // hm...
    return EXIT_SUCCESS;
}
```

```
> ./functiontemplate_infer
-1
-1
1
```

functiontemplate_infer.cc

How Do Templates Work?

```
#include "compare.h"

int main(int argc, char **argv) {
    std::cout << compare(10, 20)
    std::cout << std::endl;
    return EXIT_SUCCESS;
}

template <typename T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```

- ❖ At **compile-time**, the compiler will generate the “specialized” code from your template using the types you provided



```
int comp(const int& a, const int& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```

How Do Templates Work?

- ❖ At ***compile-time***, the compiler will generate the “specialized” code from your template using the types you provided
 - Your template definition is NOT runnable code
 - Code is *only* generated if you use your template
 - Generated machine code is specialized for the specific types of data used in the template instance (e.g.: code for `< on ints` differs from code for `< on strings`)

How Do Templates Work?

- ❖ The compiler doesn't generate any code when it sees the template function
 - It doesn't know what code to generate yet, since it doesn't know what types are involved
- ❖ When the compiler sees the function being used, then it understands what types are involved
 - It generates the ***instantiation*** of the template and compiles it (kind of like macro expansion)
 - The compiler generates template instantiations for *each* type used as a template parameter

```

compare.h: In instantiation of 'int comp(const T&, const T&) [with T =
Pair<int>]':
main.cc:10:26:   required from here
   10 |     cout << comp<Pair<int>>(Pair<int>(), Pair<int>());
       |                      ~~~~~^~~~~~
compare.h:6:9: error: no match for 'operator<' (operand types are
'const Pair<int>' and 'const Pair<int>')
    6 |     if (a < b) return -1;
       |         ~~^~~
compare.h:7:9: error: no match for 'operator<' (operand types are
'const Pair<int>' and 'const Pair<int>')
    7 |     if (b < a) return 1;
       |         ~~^~~

```

Template Instantiation Creates a Problem

```
#ifndef COMPARE_H_
#define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b);

#endif // COMPARE_H_
```

compare.h

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

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return EXIT_SUCCESS;
}
```

main.cc

```
#include "compare.h"

template <typename T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```

compare.cc

undefined reference to
`int comp<int>(int const&, int const&)

Who is responsible for
instantiating comp for ints?

Solution #1 (Google Style Guide prefers)

```
#ifndef COMPARE_H_
#define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}

#endif // COMPARE_H_
```

compare.h

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

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return EXIT_SUCCESS;
}
```

main.cc

Solution #2 (you'll see this sometimes)

```
#ifndef COMPARE_H_
#define COMPARE_H_

template <typename T>
int comp(const T& a, const T& b);

#include "compare.cc"

#endif // COMPARE_H_
```

compare.h

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

using namespace std;

int main(int argc, char **argv) {
    cout << comp<int>(10, 20);
    cout << endl;
    return EXIT_SUCCESS;
}
```

main.cc

```
template <typename T>
int comp(const T& a, const T& b) {
    if (a < b) return -1;
    if (b < a) return 1;
    return 0;
}
```

compare.cc

Pretty much the only time
you'll see a #include of
anything that's not a .h

Template Non-types

- ❖ You can use non-types (constant values) in a template:

```
#include <iostream>
#include <string>

// return pointer to new N-element heap array filled with val
// (not entirely realistic, but shows what's possible)
template <typename T, int N>
T* varray(const T &val) {
    T* a = new T[N];
    for (int i = 0; i < N; ++i)
        a[i] = val;
    return a;
}

int main(int argc, char **argv) {
    int *ip = varray<int, 10>(17);
    string *sp = varray<string, 17>("hello");
    ...
}
```

The compiler can unroll this loop!
It's a constant length in every
instantiation

Class Templates

- ❖ Templates are useful for classes as well
 - (In fact, that was one of the main motivations for templates!)
- ❖ Imagine we want a class that holds a pair of things that we can:
 - 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
 - Swap the values of the things
 - Print the pair of things

Pair Class Definition

Pair.h

```
#ifndef PAIR_H_
#define PAIR_H_

template <typename Thing> class Pair {
public:
    Pair() { };

    Thing get_first() const { return first_; }
    Thing get_second() const { return second_; }
    void set_first(const Thing &copyme);
    void set_second(const Thing &copyme);
    void Swap();

private:
    Thing first_, second_;
};

#include "Pair.cc" // or put entire template def here

#endif // PAIR_H_
```

Pair Function Definitions

Pair.cc

```
template <typename Thing>
void Pair<Thing>::set_first(const Thing &copyme) {
    first_ = copyme;
}

template <typename Thing>
void Pair<Thing>::set_second(const Thing &copyme) {
    second_ = copyme;
}

template <typename Thing>
void Pair<Thing>::Swap() {
    Thing tmp = first_;
    first_ = second_;
    second_ = tmp;
}

template <typename T>
std::ostream &operator<<(std::ostream &out, const Pair<T>& p) {
    return out << "Pair(" << p.get_first() << ", "
               << p.get_second() << ")";
}
```

Using Pair

usepair.cc

```
#include <iostream>
#include <string>

#include "Pair.h"

int main(int argc, char** argv) {
    Pair<std::string> ps;
    std::string x("foo"), y("bar");

    ps.set_first(x);
    ps.set_second(y);
    ps.Swap();
    std::cout << ps << std::endl;

    return EXIT_SUCCESS;
}
```

More Flexible Pair Class Definition

Pair.h

Can have as many
template parameters
as you want

```
#ifndef PAIR_H_
#define PAIR_H_

template <typename Thing, typename Stuff> class Pair {
public:
    Pair() { };

    Thing get_first() const { return first_; }
    Stuff get_second() const { return second_; }
    void set_first(const Thing &copyme);
    void set_second(const Stuff &copyme);
    void Swap();

private:
    Thing first_;
    Stuff second_;
};

#include "Pair.cc" // or put entire template def here

#endif // PAIR_H_
```

Class Template Notes (look in *Primer* for more)

- ❖ `Thing` is replaced with template argument when class is instantiated

- The class template parameter name is in scope of the template class definition and can be freely used there

```
Thing get_first() const { return first_; }
```

- ❖ Class template member functions are template functions with template parameters that match those of the class template

- These member functions must be defined as template function outside of the class template definition (if not written inline)

```
template <typename Thing>
void Pair<Thing>::set_first(const Thing &copyme) {
    first_ = copyme;
}
```

Lecture Outline

- ❖ Templates
- ❖ **The C++ Standard Template Library**
 - Containers: vector
 - Iterators

C++'s Standard Library

❖ C++'s Standard Library consists of four major pieces:

- 1) The entire C standard library
- 2) C++'s input/output stream library
 - `std::cin`, `std::cout`, `stringstreams`, `fstreams`, etc.
- 3) C++'s miscellaneous library
 - Strings, exceptions, memory allocation, localization
- 4) C++'s standard template library (**STL**) 
 - Containers, iterators, and algorithms (sort, find, etc.)

STL Containers 😊

- ❖ A **container** is an object that stores (in memory) a collection of other objects (elements)
 - Implemented as class templates, so hugely flexible
 - More info in *C++ Primer* §9, 11
- ❖ Several different classes of container
 - Sequence containers (`vector`, `deque`, `list`, ...)
 - Associative containers (`set`, `map`, `multiset`, `multimap`, `bitset`, ...)
 - Differ in algorithmic cost and supported operations

STL Containers

- ❖ STL containers store by *value*, not by *reference*
 - When you insert an object, the container makes a *copy*
 - If the container needs to rearrange objects, it makes copies
 - *e.g.* if you sort a `vector`, it will make many, many copies
 - *e.g.* if you insert into a `map`, that may trigger several copies
 - What if you don't want this (disabled copy constructor or copying is expensive)?
 - You can insert a pointer
 - You can insert a wrapper object with a pointer to the object
 - We'll learn about these “smart pointers” soon

STL **vector**

- ❖ A generic, dynamically resizable array
 - <http://www.cplusplus.com/reference/stl/vector/vector/>
 - Elements are stored in *contiguous* memory locations
 - Elements can be accessed using pointer arithmetic if you'd like
 - Random access is $O(1)$ time
 - Adding/removing from the end is cheap (amortized constant time)
 - Inserting/deleting from the middle or start is expensive (linear time)

Tracer Class

- ❖ Wrapper class for an `int` `value_`
 - Class and member definitions can be found in `Tracer.h` and `Tracer.cc`
- ❖ Useful for tracing behaviors of containers
 - All methods print identifying messages
 - Unique `id_` allows you to follow individual instances

Tracer Example Walkthrough

See:

`Tracer.h`

`Tracer.cc`

`vectorfun.cc`

<https://courses.cs.washington.edu/courses/cse333/25su/lecture/13-c++-templates+STL-example/>

Why All the Copying?

- ❖ What's going on here?
- ❖ Answer: a C++ vector (like Java's ArrayList) is initially small, but grows if needed as elements are added
 - Implemented by allocating a new, larger underlying array, copy existing elements to new array, and then replace previous array with new one
- ❖ And vector starts out *really* small by default, so it needs to grow almost immediately!
 - But you can specify an initial capacity if “really small” is an inefficient initial size (use “reserve” member function)
 - Example: see vectorcap.cc

Lecture Outline

- ❖ Templates
- ❖ **The C++ Standard Template Library**
 - Containers: vector
 - **Iterators**

STL `iterator`

- ❖ Each container class has an associated `iterator` class (*e.g.* `vector<int>::iterator`) used to iterate through elements of the container
 - <http://www.cplusplus.com/reference/std/iterator/>
 - `Iterator range` is from `.begin()` up to `.end()`
 - `end` is one past the last container element!
 - Some container iterators support more operations than others
 - All can be incremented (`++`), copied, copy-constructed
 - Some can be dereferenced on RHS (*e.g.* `x = *it;`)
 - Some can be dereferenced on LHS (*e.g.* `*it = x;`)
 - Some can be decremented (`--`)
 - Some support more (`[]`, `+`, `-`, `+=`, `-=`, `<`, `>` operators)

iterator Example

vectoriterator.cc

```
#include <vector>

#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    vector<Tracer>::iterator it;
    for (it = vec.begin(); it < vec.end(); it++) {
        cout << *it << endl;
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

Type Inference (C++11)

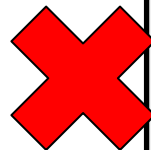
- ❖ The `auto` keyword can be used to infer types
 - Simplifies your life if, for example, functions return complicated types
 - The expression using `auto` must contain explicit initialization for it to work

```
// Calculate and return a vector
// containing all factors of n
std::vector<int> Factors(int n);

void foo(void) {
    // Manually identified type
    std::vector<int> facts1 =
        Factors(324234);

    // Inferred type
    auto facts2 = Factors(12321);

    // Compiler error here
    auto facts3;
}
```



auto and Iterators

- ❖ Life becomes much simpler!

```
for (vector<Tracer>::iterator it = vec.begin(); it < vec.end(); it++) {  
    cout << *it << endl;  
}
```



```
for (auto it = vec.begin(); it < vec.end(); it++) {  
    cout << *it << endl;  
}
```

Range for Statement (C++11)

- ❖ Syntactic sugar similar to Java's `foreach`

```
for ( declaration : expression ) {  
    statements  
}
```

- *declaration* defines loop variable
- *expression* is an object representing a sequence
 - Strings, initializer lists, arrays with an explicit length, `std::vector`, `std::array`, and other containers that support iterators

```
// Prints out a string, one  
// character per line  
std::string str("hello");  
  
for ( auto c : str ) {  
    std::cout << c << std::endl;  
}
```

Copy of the
character

Updated iterator Example

vectoriterator_2011.cc

```
#include <vector>

#include "Tracer.h"

using namespace std;

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(a);
    vec.push_back(b);
    vec.push_back(c);

    cout << "Iterating:" << endl;
    for (auto & p : vec) {    // p is a reference (alias) of vec
        cout << p << endl;    // element here; not a new copy
    }
    cout << "Done iterating!" << endl;
    return EXIT_SUCCESS;
}
```

STL Algorithms

- ❖ A set of functions to be used on ranges of elements
 - **Range**: any sequence that can be accessed through *iterators* or *pointers*, like arrays or some of the containers
 - General form: `algorithm(begin, end, ...);`
- ❖ Algorithms operate directly on range *elements* rather than the containers they live in
 - Make use of elements' copy ctor, =, ==, !=, <
 - Some do not modify elements
 - e.g. find, count, for_each, min_element, binary_search
 - Some do modify elements
 - e.g. sort, transform, copy, swap

Algorithms Example

vectoralgos.cc

```
#include <vector>
#include <algorithm>
#include "Tracer.h"
using namespace std;

void PrintOut(const Tracer& p) {
    cout << " printout: " << p << endl;
}

int main(int argc, char** argv) {
    Tracer a, b, c;
    vector<Tracer> vec;

    vec.push_back(c);
    vec.push_back(a);
    vec.push_back(b);
    cout << "sort:" << endl;
    sort(vec.begin(), vec.end());
    cout << "done sort!" << endl;
    for_each(vec.begin(), vec.end(), &PrintOut);
    return EXIT_SUCCESS;
}
```

Administrivia

- ❖ Homework 2 due **tomorrow night** (7/18)
 - Don't forget to clone your repo to double-/triple-/quadruple-check compilation, execution, and tests!

Review Questions (both template and class issues)

- ❖ Why are only `get_first()` and `get_second()` `const`?
- ❖ Why do the accessor methods return `Thing` and not references?
- ❖ Why is `operator<<` not a friend function?
- ❖ What happens in the default constructor when `Thing` is a class?
- ❖ In the execution of `Swap()`, how many times are each of the following invoked (assuming `Thing` is a class)?
ctor _____ cctor _____ op= _____ dtor _____

Extra Exercise #1

- ❖ Using the `Tracer.h/.cc` files from lecture:
 - Construct a vector of lists of Tracers
 - *i.e.* a `vector` container with each element being a `list` of Tracers
 - Observe how many copies happen 😊
 - Use the sort algorithm to sort the vector
 - Use the `list.sort()` function to sort each list