

# C++ Templates

## CSE 333 Spring 2019

**Instructor:** Justin Hsia

**Teaching Assistants:**

Aaron Johnston

Andrew Hu

Daniel Snitkovskiy

Forrest Timour

Kevin Bi

Kory Watson

Pat Kosakanchit

Renshu Gu

Tarkan Al-Kazily

Travis McGaha

# Administrivia

- ❖ Exercise 11 released today, due Wednesday
  - Modify your Vector class to use the heap
  - Refer to `Str.h/Str.cc`
- ❖ Homework 2 due Thursday (5/2)
  - File system crawler, indexer, and search engine
  - Don't forget to clone your repo to double-/triple-/quadruple-check compilation!
- ❖ Midterm: next Friday (5/10) from 5-6:10 pm in KNE 130
  - Let me know ASAP if you have a conflict!

# Lecture Outline

## ❖ Templates

# Suppose that...

- ❖ You want to write a function to compare two ints
- ❖ You want to write a function to compare two strings
  - 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;  
}
```

does something different in each case

```
// returns 0 if equal, 1 if value1 is bigger, -1 otherwise  
int compare(const string &value1, const string &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 *too much repeated code!*
- ❖ What we'd prefer to do is write "*generic code*"
  - Code that is **type-independent**
  - Code that is **compile-type polymorphic** across types

# 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 or values as arguments to it
- ★ 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

# 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> // <...> can also be written <class T>
int compare(const T &value1, const T &value2) {
    if (value1 < value2) return -1;
    if (value2 < value1) return 1;
    return 0;
}

int main(int argc, char **argv) {
    std::string h("hello"), w("world");
    std::cout << compare<int>(10, 20) << std::endl; // -1
    std::cout << compare<std::string>(h, w) << std::endl; // -1
    std::cout << compare<double>(50.5, 50.6) << std::endl; // -1
    return EXIT_SUCCESS;
}
```

template  
function  
definition

template parameter list (also written as: <class T>)

T only needs to implement < to work with compare()

explicit template arguments  
(3 different instances of compare)

functiontemplate.cc

# Compiler Inference

- ❖ Same thing, 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::string h("hello"), w("world");
    std::cout << compare(10, 20) << std::endl; // ok infer int, -1
    std::cout << compare(h, w) << std::endl; // ok infer std::string, -1
    std::cout << compare("Hello", "World") << std::endl; // hm...
    return EXIT_SUCCESS; // infer char*, could be ±1 (based on addresses)
}
```

functiontemplate\_infer.cc



# 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* valarray(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 = valarray<int, 10>(17);
    string *sp = valarray<string, 17>("hello");
    ...
}
```

↑ separate template arguments with commas like normal

# What's Going On?

- ❖ 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

# This 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

*g++ -c compare.cc → empty compare.o!*

```
#include "compare.h" (no usage of comp< >)

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

*g++ -c main.cc → main.o  
without  
definition of  
comp<int>*

*g++ main.o compare.o → linker error  
(no comp<int>)*

# 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

less implementation hiding! ☺

# 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

```
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

```
#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

# Practice Question

- ❖ Assume we are using Solution #2 (.h includes .cc)
- ❖ Which is the best way to compile our program (a.out)?
  - Vote at <http://PollEv.com/justinh>

A. `g++ main.cc`

B. `g++ main.cc compare.cc`

C. `g++ main.cc compare.h`

D. `g++ -c main.cc`

~~`g++ -c compare.cc`~~ → empty compare.o

~~`g++ main.o compare.o`~~

E. We're lost...

template definition added to main.cc  
via #include by preprocessor  
(main.cc → compare.h → compare.cc)

all of these command sets will work,  
but crossed out parts are  
unnecessary.

Make file

```
a.out: main.cc compare.h compare.cc
g++ main.cc
```

# 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() { }; //default constructor

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

private:
    Thing first_, second_;
};

#include "Pair.cc" ← following Solution #2

#endif // _PAIR_H_
```

*template class definition*

*//default constructor*

*} inline definitions*

*} declarations*

*could be primitive or another class*



# Pair Function Definitions

Pair.cc

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

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

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

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

template function for  
template class member function

member of template class instantiation with Thing

# Using Pair

usepair.cc

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

#include "Pair.h"

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

    ps.set_first(x); // ("foo", "")
    ps.set_second(y); // ("foo", "bar")
    ps.Swap(); // ("bar", "foo")
    std::cout << ps << std::endl; // invoke nonmember operator<< function

    return EXIT_SUCCESS;
}
```

# 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
  - 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)
      - The template parameter name does *not* need to match that used in the template class definition, but really should
  - Only template methods that are actually called in your program are instantiated (but this is an implementation detail)

# Review Questions (Classes and Templates)

- ❖ Why are only `get_first()` and `get_second()` const?  
*the accessors don't modify the class instance — the mutators and swap do  
 — operator << is a non-member function*
- ❖ Why do the accessor methods return `Thing` and not references?  
*returning a reference to a private member violates the 'private' modifier,  
 so we instead return a copy of Thing*
- ❖ Why is `operator<<` not a `friend` function?  
*it doesn't need access to private data members because it uses the  
 accessors instead*
- ❖ What happens in the default constructor when `Thing` is a class?  
*data members still get initialized — in this case, invoke the default constructor  
 of Thing for first\_ and second\_*
- ❖ In the execution of `Swap()`, how many times are each of the  
 following invoked (assuming `Thing` is a class)?

ctor 0cctor 1  
tmpop= 2  
first\_, second\_dctor 1  
tmp