C++ Templates
CSE 333 Fall 2023

Instructor: Chris Thachuk

Teaching Assistants:

Ann Baturytski   Humza Lala

Yuquan Deng   Alan Li

Noa Ferman   Leanna Mi Nguyen

James Froelich   Chanh Truong

Hannah Jiang   Jennifer Xu

Yegor Kuznetsov

Relevant Course Information

❖ Midterm grades out later this week
  ▪ More feedback after grading is complete

❖ Exercise 7 due Wednesday (11/1)

❖ Exercise 8 will go out early
  ▪ For those ready to begin
  ▪ *Note*: STL won't be covered until Wednesday's lecture

❖ Homework 2 due ~~(10/30)~~ tomorrow (10/31) by 10pm
  ▪ Don't forget to clone your repo to double-/triple-/quadruple-check compilation!

Tagging Your Homework Correctly

❖ Purpose of Homework 0 is to understand this process

❖ Leniency on Homework 1 if you failed to tag properly

- If you did not tag your homework 1 correctly then:

  1. Properly tag a commit that was pushed to the server by the deadline, and push that tag to gitlab server.

  2. Open a regrade request explaining you incorrectly tagged your Hw1 and confirm you corrected the tag on the server.

❖ <u>No exceptions going forward on tagging</u>

Lecture Outline

❖ **Templates**

Suppose that...

- ❖ You want to write a function to compare two `int`s
- ❖ You want to write a function to compare two `string`s
    - ▪ Function overloading!

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

Hmm…

❖ 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":

```cpp
#include <iostream>
#include <string>
                        template parameter list (also written as: <class T>)
// 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;
}
         T only needs to implement  < to work with compare()

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

explicit template arguments
(3 different instances of compare)

functiontemplate.cc

8

Compiler Inference

❖ Same thing, but letting the compiler infer the types:

```cpp
#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)
```

— no explicit template argument

functiontemplate_infer.cc

Template Non-types

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

```cpp
#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

valtemplate.cc

UNIVERSITY *of* WASHINGTON

What's Going On?

❖ The compiler <u>doesn't generate any code when it sees the template function definition</u>

  ■ It doesn't know what code to generate yet, since it doesn't know what types are involved   *Different behavior for different types*

❖ <u>When the compiler sees the function being used</u>, 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)

```cpp
#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

```cpp
#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)

```cpp
#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

```cpp
#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

```cpp
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

14

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: *Pair*

  ▪ 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_; }    // inline definitions
  void  set_first(const Thing& copyme);
  void  set_second(const Thing& copyme);    // declarations
  void  Swap();

 private:
  Thing first_, second_;    // could be primitive or another class
};

#include "Pair.cc"    // following Solution #2 (for ease of slide separation)

#endif  // PAIR_H_
```

template class definition (annotation)

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;
}
// 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() << ")";
}
```

(handwritten annotations:) template function for template class member function

member of template class instantiation with Thing

Using Pair

usepair.cc

```cpp
#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 __0__   cctor __1__   op= __2__   dtor __1__

tmp   first_, second_   tmp