

C++ Templates

CSE 333 Autumn 2025

Instructors: Naomi Alterman, Chris Thachuk

Teaching Assistants:

Ann Baturytski

Derek de Leuw

Blake Diaz

Rishabh Jain

Chendur Jel Jayavelu

Lucas Kwan

Irene Xin Jie Lau

Nathan Li

Maya Odenheim

Advay Patil

Selim Saridede

Deeksha Vatwani

Angela Wu

Jiexiao Xu

Administrivia

- ❖ Homework 2 due tomorrow
 - File system crawler, indexer, and search engine
 - Don't forget to clone your repo to double-/triple-/quadruple-check compilation, execution, and tests!
- ❖ Ex10 out, due 10/29 (next Wednesday, after midterm 😊)
- ❖ HW1 feedback notes
 - If you have questions or spot problems use a gradescope regrade request to ask (not ed, not email). That will route it to the right person
 - “Does-it-work” and “code quality” are two different scores with two max point values. The scales are not the same, and we don't just add the numbers together to get your grade (though it looks that way in Gradescope)

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;
}
```

```
// 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
- ❖ 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
 - 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)

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;
    std::cout << compare<std::string>(h, w) << std::endl;
    std::cout << compare<double>(50.5, 50.6) << std::endl;
    return EXIT_SUCCESS;
}
```

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
    std::cout << compare(h, w) << std::endl;    // ok
    std::cout << compare("Hello", "World") << std::endl; // hm...
    return EXIT_SUCCESS;
}
```

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
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");
    ...
}
```

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

```
#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

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

pollev.com/naomila

**Think about #define macros and C++ templates.
How are they similar? How are they different?**



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(Thing &copyme);
    void set_second(Thing &copyme);
    void Swap();

private:
    Thing first_, second_;
};

#include "Pair.cc" // or (better?) put entire template def here

#endif // _PAIR_H_
```

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;
}

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;
}
```

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 functions 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 to minimize reader confusion
 - Only template methods that are actually called in your program are instantiated (but this is an implementation detail)

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 _____

End of midterm content!

