HTTP & Templates

Computer Systems Programming, Spring 2024

Instructor: Travis McGaha

TAs:

Ash Fujiyama
CV Kunjeti
Felix Sun
Heyi Liu
Kevin Bernat
Lang Qin
Sean Chuang
Serena Chen
Yuna Shao

Logistics

- HW2 Posted Late Due Wed 3/27 @ 11:59
  - Auto-grader posted
  - We know it is not working for some, we are investigating
- Exam grades to be posted this week
- HW03 to be released sometime this week

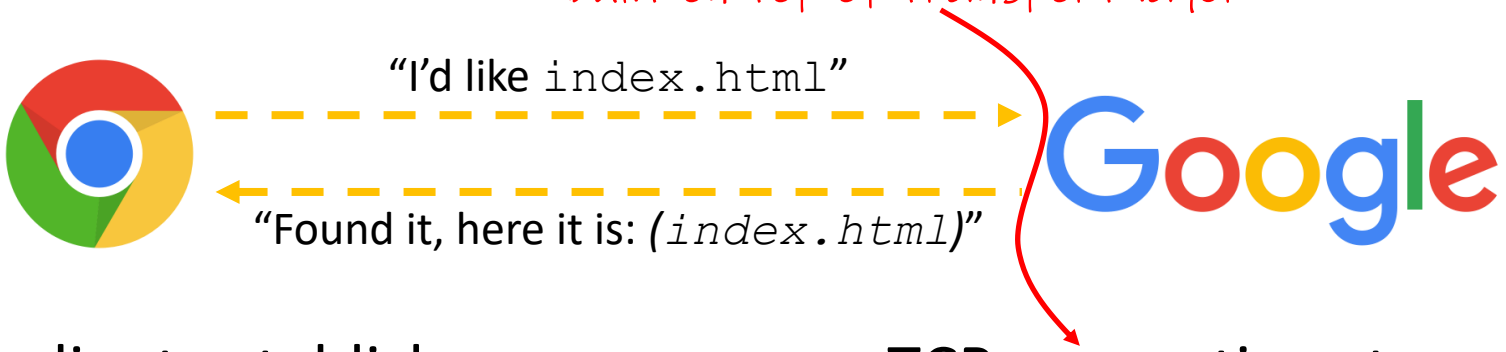
pollev.com/tqm

- Any questions before we begin?

Lecture Outline

- **HTTP**
- Templates

HTTP Basics



- A client establishes one or more TCP connections to a server
  - The client sends a request for a web object over a connection and the server replies with the object's contents
- We have to figure out how to let the client and server communicate their intentions to each other clearly
  - We have to define a *protocol*

Protocols

- A protocol is a set of rules governing the format and exchange of messages in a computing system
  - What messages can a client exchange with a server?
    - What is the syntax of a message?
    - What do the messages mean?
    - What are legal replies to a message?
  - What sequence of messages are legal?
    - How are errors conveyed?

- A protocol is (roughly) the network equivalent of an API

HTTP

- Hypertext Transport Protocol
  - A request / response protocol
    - A client (web browser) sends a request to a web server
    - The server processes the request and sends a response
  - Typically, a **request** asks a server to retrieve a resource
    - A *resource* is an object or document, named by a Uniform Resource Identifier (**URI**)
  - A **response** indicates whether or not the server succeeded
    - If so, it provides the content of the requested response
  - More info: https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol

e.g. a webpage, image, etc

HTTP Requests

- General form:

```
[METHOD] [request-uri] HTTP/[version]\r\n
[headerfield1]: [fieldvalue1]\r\n
[headerfield2]: [fieldvalue2]\r\n
[...]
[headerfieldN]: [fieldvalueN]\r\n
\r\n
[request body, if any]
```

Type of Action to take

Resource to act on

In this class, 1.1

Any# of headers (designed for flexibility)

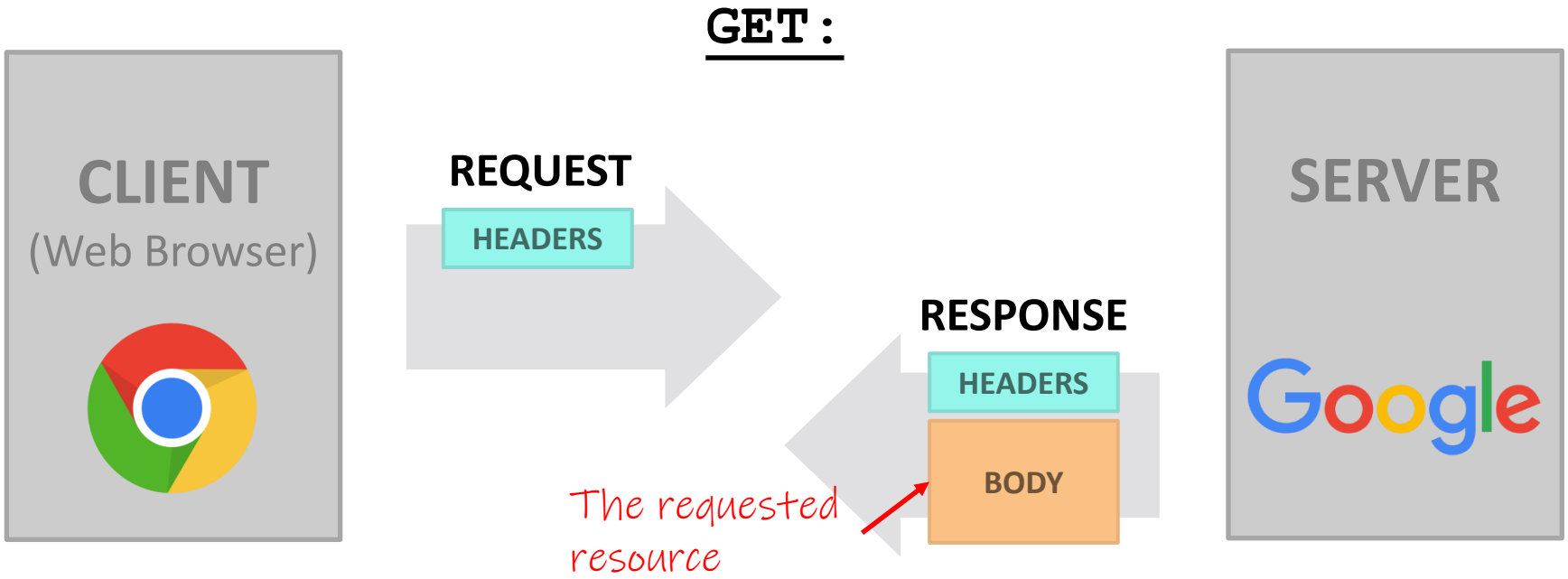
Blank line to indicate the end of the headers.

\r\n is used to indicate a "new line" in HTTP

HTTP Methods

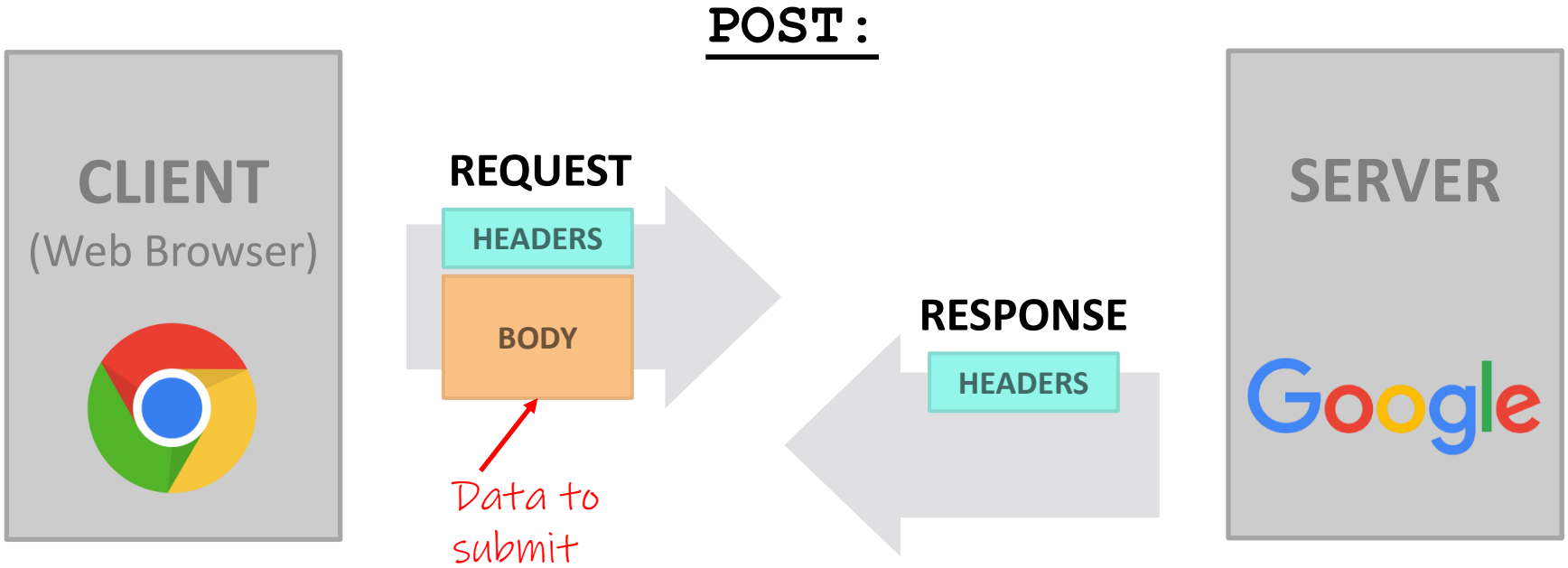
- There are three commonly-used HTTP methods:
  - **GET**: "Please send me the named resource"

Used in the project



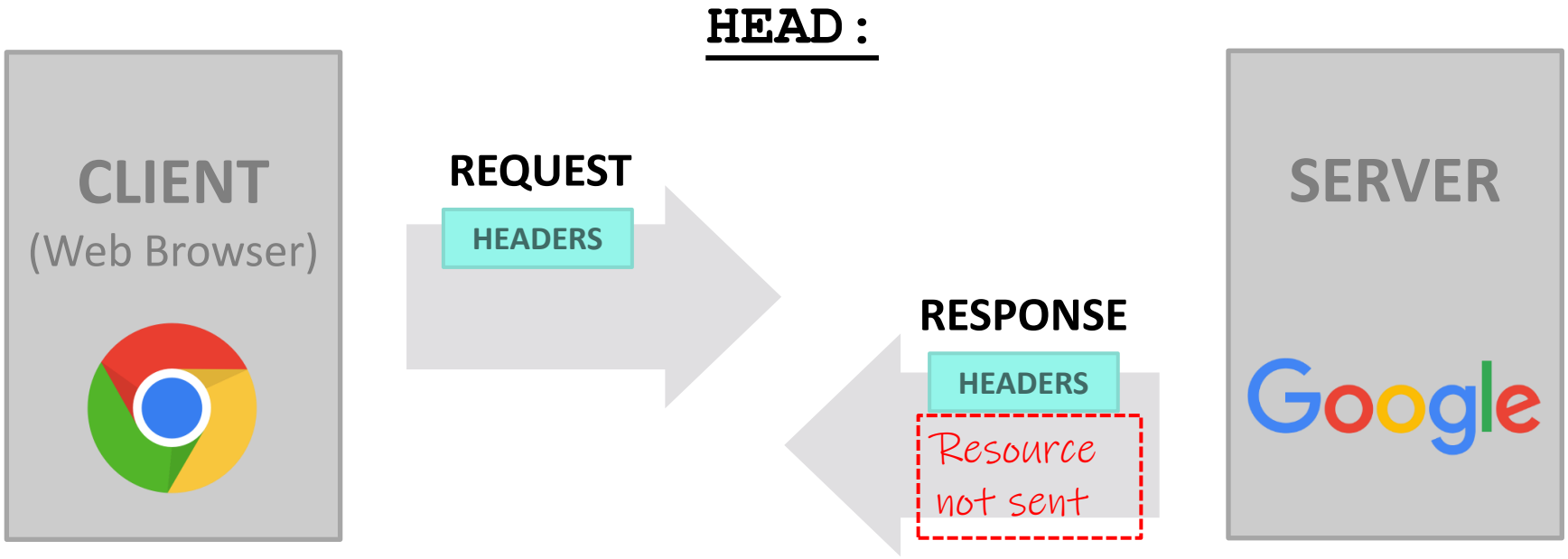
HTTP Methods

- There are three commonly-used HTTP methods:
  - `GET`: "Please send me the named resource"
  - **`POST`**: "I'd like to submit data to you" (*e.g.* file upload)



HTTP Methods

- There are three commonly-used HTTP methods:
  - `GET`: "Please send me the named resource"
  - `POST`: "I'd like to submit data to you" (*e.g.* file upload)
  - **`HEAD`**: "Send me the headers for the named resource"
    - Doesn't send resource; often to check if cached copy is still valid



HTTP Methods

- There are three commonly-used HTTP methods:
  - `GET`: "Please send me the named resource"
  - `POST`: "I'd like to submit data to you" (*e.g.* file upload)
  - `HEAD`: "Send me the headers for the named resource"
    - Doesn't send resource; often to check if cached copy is still valid

- Other methods exist, but are much less common:
  - `PUT`, `DELETE`, `TRACE`, `OPTIONS`, `CONNECT`, `PATCH`, `...`
    - For instance: `TRACE` – "show any proxies or caches in between me and the server"

HTTP Uniform Resource Identifier (URI)

- Absolute URI
  - Composition: `scheme:[//authority]path[?query]`
  - Mainly used for communicating via proxy

- Most common form of Request-URI
  - Composition: `path[?query]`
  - Host is specified through headers
  - Query is optional
  - Path can be empty (just `/`)

- Example Request-URI:
  - /static/test_tree/books/artofwar.txt?terms=hello

HTTP Versions

- All current browsers and servers "speak" HTTP/1.1
  - Version 1.1 of the HTTP protocol
    - https://www.w3.org/Protocols/rfc2616/rfc2616.html
  - Standardized in 1997 and meant to fix shortcomings of HTTP/1.0
    - Better performance, richer caching features, better support for multihomed servers, and much more

- HTTP/2 standardized recently (published in 2015)
  - Allows for higher performance but doesn't change the basic web request/response model
  - Will coexist with HTTP/1.1 for a long time

Hard to change/force a switch in the "wild"

Client Headers

- The client can provide one or more request "headers"
  - These provide information to the server or modify how the server should process the request

- You'll encounter many in practice
  - https://www.w3.org/Protocols/rfc2616/rfc2616-sec5.html
  - `Host:` the DNS name of the server <- server my host multiple domains
  - `User-Agent:` an identifying string naming the browser Desktop vs. mobile
  - `Accept:` the content types the client prefers or can accept
  - `Cookie:` an HTTP cookie previously set by the server

A Real Request

request uri version

```
GET / HTTP/1.1
Host: breadsouth-turbopromo.codio.io:3333
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/66.0.3359.181 Safari/537.36
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,
image/apng,*/*;q=0.8
DNT: 1
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.9
Cookie: SESS0c8e598bbe17200b27e1d0a18f9a42bb=5c18d7ed6d369d56b69a1c0aa441d7
8f; SESSd47cbe79be51e625cab059451de75072=d137dbe7bbe1e90149797dcd89c639b1;
_sdsat_DMC_or_CCODE=null; _sdsat_utm_source=; _sdsat_utm_medium=; _sdsat_ut
m_term=; _sdsat_utm_content=; adblock=blocked; s_fid=50771A3AC73B3FFF-3F18A
ABD559FFB5D; s_cc=true; prev_page=science.%3A%2Fcontent%2F347%2F6219%2F262%
2Ftab-pdf; ist_usr_page=1; sat_ppv=79; ajs_anonymous_id=%229225b8cf-6637-49
c8-8568-ecb53cfc760c%22; ajs_user_id=null; ajs_group_id=null; __utma=598078
07.316184303.1491952757.1496310296.1496310296.1; __utmc=59807807; __utmc=80
...
```

Keep connection alive after this request

Chrome windows desktop

HTTP Responses

- General form:
  - `HTTP/[version] [status code] [reason]\r\n`
    `[headerfield1]: [fieldvalue1]\r\n`
    `[headerfield2]: [fieldvalue2]\r\n`
    `[...]`
    `[headerfieldN]: [fieldvalueN]\r\n`
    `\r\n`
    `[response body, if any]`

A number

A Human readable string

Typically the requested resource

Status Codes and Reason

- *Code*: numeric outcome of the request – easy for computers to interpret
  - A 3-digit integer with the 1st digit indicating a response category
    - `1xx`: Informational message
    - `2xx`: Success
    - `3xx`: Redirect to a different URL
    - `4xx`: Error in the client's request
    - `5xx`: Error experienced by the server
- *Reason*: human-readable explanation
  - *e.g.* "OK" or "Moved Temporarily"

Common Statuses

- `HTTP/1.1 200 OK`
  - The request succeeded and the requested object is sent
- `HTTP/1.1 404 Not Found`
  - The requested object was not found
- `HTTP/1.1 301 Moved Permanently`
  - The object exists, but its name has changed
    - The new URL is given as the "`Location:`" header value
- `HTTP/1.1 500 Server Error`
  - The server had some kind of unexpected error

Server Headers

- The server can provide zero or more response "headers"
  - These provide information to the client or modify how the client should process the response

- You'll encounter many in practice
  - https://www.w3.org/Protocols/rfc2616/rfc2616-sec6.html
  - `Server:` a string identifying the server software
  - `Content-Type:` the type of the requested object
  - `Content-Length:` size of requested object
  - `Last-Modified:` a date indicating the last time the request object was modified

How to interpret resource (image, text…)

When to stop reading

A Real Response

version status reason

```
HTTP/1.1 200 OK
Date: Mon, 21 May 2018 07:58:46 GMT
Server: Apache/2.2.32 (Unix) mod_ssl/2.2.32 OpenSSL/1.0.1e-fips
mod_pubcookie/3.3.4a mod_uwa/3.2.1 Phusion_Passenger/3.0.11
Last-Modified: Mon, 21 May 2018 07:58:05 GMT
ETag: "2299e1ef-52-56cb2a9615625"
Accept-Ranges: bytes
Content-Length: 82
Vary: Accept-Encoding,User-Agent
Connection: close
Content-Type: text/html
Set-Cookie:
bbbbbbbbbbbbbbb=DBMLFDMJCGAOILMBPIIAAIFLGBAKOJNNMCJIKKBKCDMDEJHMPONHCILPIBL
ADEAKCIABMEEPAOPMMKAOLHOKJMIGMIDKIHNCANAPHMFMBLBABPFENPDANJAPIBOIOOOD;
HttpOnly

<html><body>
<font color="chartreuse" size="18pt">Awesome!!</font>
</body></html>
```

Length of response body

Close connection after transaction

response body is the requested html page

Cool HTTP/1.1 Features

This is extra (non-testable) material

- "Chunked Transfer-Encoding"
  - A server might not know how big a response object is
    - *e.g.* dynamically-generated content in response to a query or other user input
  - How do you send Content-Length?
    - Could wait until you've finished generating the response, but that's not great in terms of *latency* – we want to start sending the response right away
  - Chunked message body: response is a series of chunks

Cool HTTP/1.1 Features

This is extra (non-testable) material

- Persistent connections
  - Establishing a TCP connection is costly
    - Multiple network round trips to set up the TCP connection
    - TCP has a feature called "slow start"; slowly grows the rate at which a TCP connection transmits to avoid overwhelming networks
  - A web page consists of multiple objects and a client probably visits several pages on the same server
    - Bad idea: separate TCP connection for each object
    - Better idea: single TCP connection, multiple requests

20 years later…

This is extra (non-testable) material

- World has changed since HTTP/1.1 was adopted
  - Web pages were a few hundred KB with a few dozen objects on each page, now several MB each with hundreds of objects (JS, graphics, …) & multiple domains per page
  - Much larger ecosystem of devices (phones especially)
  - Many hacks used to make HTTP/1.1 performance tolerable
    - Multiple TCP sockets from browser to server
    - Caching tricks; JS/CSS ordering and loading tricks; cookie hacks
    - Compression/image optimizations; splitting/sharding requests
    - etc., etc. …

HTTP/2

This is extra (non-testable) material

- Based on Google SPDY; standardized in 2015
  - Binary protocol - easier parsing by machines (harder for humans); sizes in headers, not discovered as requests are processed; …
    - But same core request/response model (GET, POST, OK, …)
  - Multiple data steams multiplexed on single TCP connections
  - Header compression, server push, object priorities, more…
- All existing implementations incorporate TLS encryption (https)
- Supported by all major browsers and servers since ~2015
- Used now by most major web sites
  - Coexists with HTTP/1.1
  - HTTP/2 used automatically when browser and server both support it

pollev.com/tqm

- Are the following statements True or False?

	Q1	Q2
A.	False	False
B.	False	True
C.	True	False
D.	True	True
E.	We're lost…	

Q1: A protocol only defines the "syntax" that clients and servers can communicate with.

Q2: Clients and servers use the same header fields.

Poll Everywhere

pollev.com/tqm

- Are the following statements True or False?

	Q1	Q2
A.	False	False
B.	False	True
C.	True	False
D.	True	True
E.	We're lost…	

Q1: A protocol only defines the "syntax" that clients and servers can communicate with. Also the semantics/meaning

Q2: Clients and servers use the same header fields.

Poll Everywhere

pollev.com/tqm

- Which HTTP status code family do you think the following Reasons belong to?

	Q1	Q2
A.	4xx	2xx
B.	4xx	3xx
C.	5xx	2xx
D.	5xx	3xx
E.	We're lost…	

Q1: Gateway Time-out

Q2: No Content

Poll Everywhere

pollev.com/tqm

- Which HTTP status code family do you think the following Reasons belong to?

1xx: info
2xx: success
3xx: redirect
4xx: client fail
5xx: server fail

	Q1	Q2
A.	4xx	2xx
B.	4xx	3xx
C.	5xx	2xx
D.	5xx	3xx
E.	We're lost…	

Q1: Gateway Time-out

Server acting as gateway timed out

Q2: No Content

Ok! Resource retrieved, but it is empty

Lecture Outline

- HTTP
- **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!

```
// 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;
}
```

does something different in each case

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

Template parameter list

Only uses operator< to minimize requirements on T

Explicit type argument

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

Infers int

Infers string

No type specified

Infers char*? Does address integer comparison ☹

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");
  ...
}
```

Fixed type template parameter

Use comma separated list to specify template arguments

valtemplate.cpp

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

```hpp
#ifndef COMPARE_HPP_
#define COMPARE_HPP_

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

#endif  // COMPARE_HPP_
```

compare.hpp

```cpp
#include "compare.hpp"

using namespace std;

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

main.cpp

```cpp
#include "compare.hpp"

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

compare.cpp

Steps to compile

g++ -c compare.cpp

Creates an empty .o file since comp<>() is not used!

g++ -c main.cpp

No comp<int> definition, expects it to be linked in later

g++ -o main main.o compare.o

No comp<int> definition, compiler error!

Solution #1 (Google Style Guide prefers)

```
#ifndef COMPARE_HPP_
#define COMPARE_HPP_

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.hpp

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

using namespace std;

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

main.cpp

Doesn't hide implementation ☹

Solution #2 (you'll see this sometimes)

```
#ifndef COMPARE_HPP_
#define COMPARE_HPP_

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

#include "compare.cpp"

#endif // COMPARE_HPP_
```

compare.hpp

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

using namespace std;

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

main.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.cpp

Poll Everywhere

pollev.com/tqm

- Assume we are using Solution #2 (`.h` includes `.cc`)
- Which is the *simplest* way to compile our program (`a.out`)?

A. `g++ main.cpp`

B. `g++ main.cpp compare.cpp`

C. `g++ main.cpp compare.hpp`

D.
```
g++ -c main.cpp
g++ -c compare.cpp
g++ main.o compare.o
```

E. We're lost…

Poll Everywhere

pollev.com/tqm

- Assume we are using Solution #2 (`.h` includes `.cpp`)
- Which is the *simplest* way to compile our program (`a.out`)?

A. `g++ main.cpp`

B. `g++ main.cpp compare.cpp`

C. `g++ main.cpp compare.hpp`

Template definition added via #include"compare,h"

D. `g++ -c main.cpp`
`g++ -c compare.cpp` → Empty object file
`g++ main.o compare.o`

E. We're lost…

All of the commands will work, but crossed out parts are unnecessary.

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.hpp

```
#ifndef PAIR_HPP_
#define PAIR_HPP_

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.cpp"

#endif  // PAIR_HPP_
```

Template parameters for class definition

Could be objects, could be primitives

Using solution #2

Pair Function Definitions

Pair.cpp

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

Definition of Member function of template class

Member of template class

Non member function to print out data in template class discussed later in semester

Using Pair

usepair.cpp

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

#include "Pair.hpp"

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;
}
```

Invokes default ctor, which default constructs members ("", "")

("foo", "")

("foo", "bar")

("bar", "foo")

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)