The final quarter

Roadmap

- Q1 : Basic programming using functions
- Q2 : Pointers and low-level programming
- Q3 : Objects and high-level programming
- Q4 : Designing and delivering software
  - Designing APIs and interfaces
  - Collaborative coding
  - Selecting and using existing containers and algorithms
  - Implementing algorithms
  - Error handling
  - Debugging

Lectures...

- From next week lecture timetable changes a bit
  - Tue at 9:00 -> Mon at 10:00
- If the strikes are on, then I won't lecture
  - Upcoming strike days cover Thu 20th and Mon 24th

- Lab should be unaffected (by me)
  - Materials still released (tomorrow)
  - Content should still make sense

Some hidden skills you now have

- The portfolios are dual purpose
  1. Support learning of C++ constructs
  2. Develop skills needed for software engineering
- Some of the skills you now have
  - Reading and implementing specifications
  - Delivering programs that can be deployed
  - Creating a program with well-defined IO
  - Refactoring programs while keeping them working
  - Using scripts to automate build and test

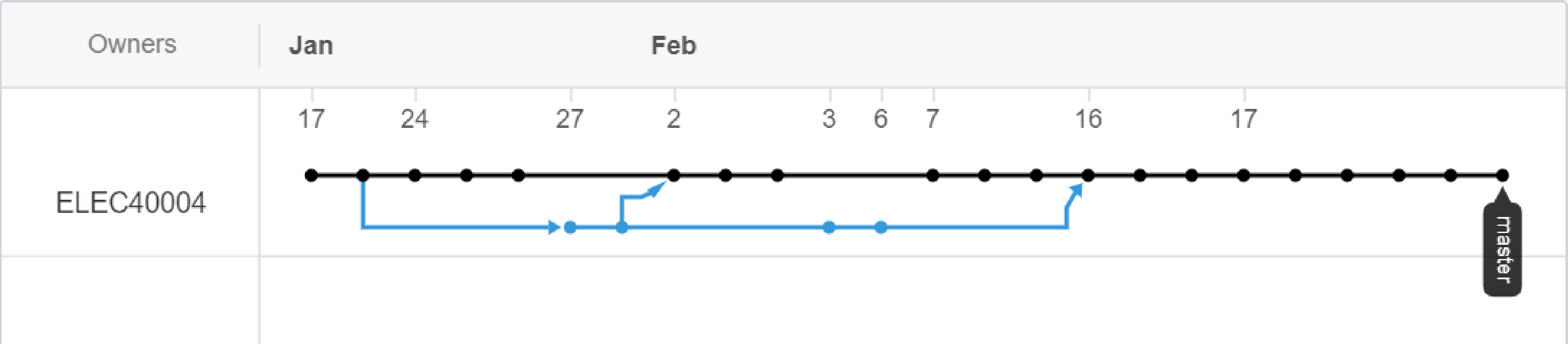
This is engineering : *delivering things that work*

A skill to develop : collaboration

- Engineering is inherently collaborative
  - It is ***very*** rare for one person to work alone
  - Academic research (PhDs) are the main exception
- We need ways to collaborate effectively
  - Sharing knowledge
  - Splitting up systems
  - Defining interfaces
  - Allowing group members to work independently
  - Integrating work into a coherent whole
- Tools can play a big role here

Git – wasn't that fun!

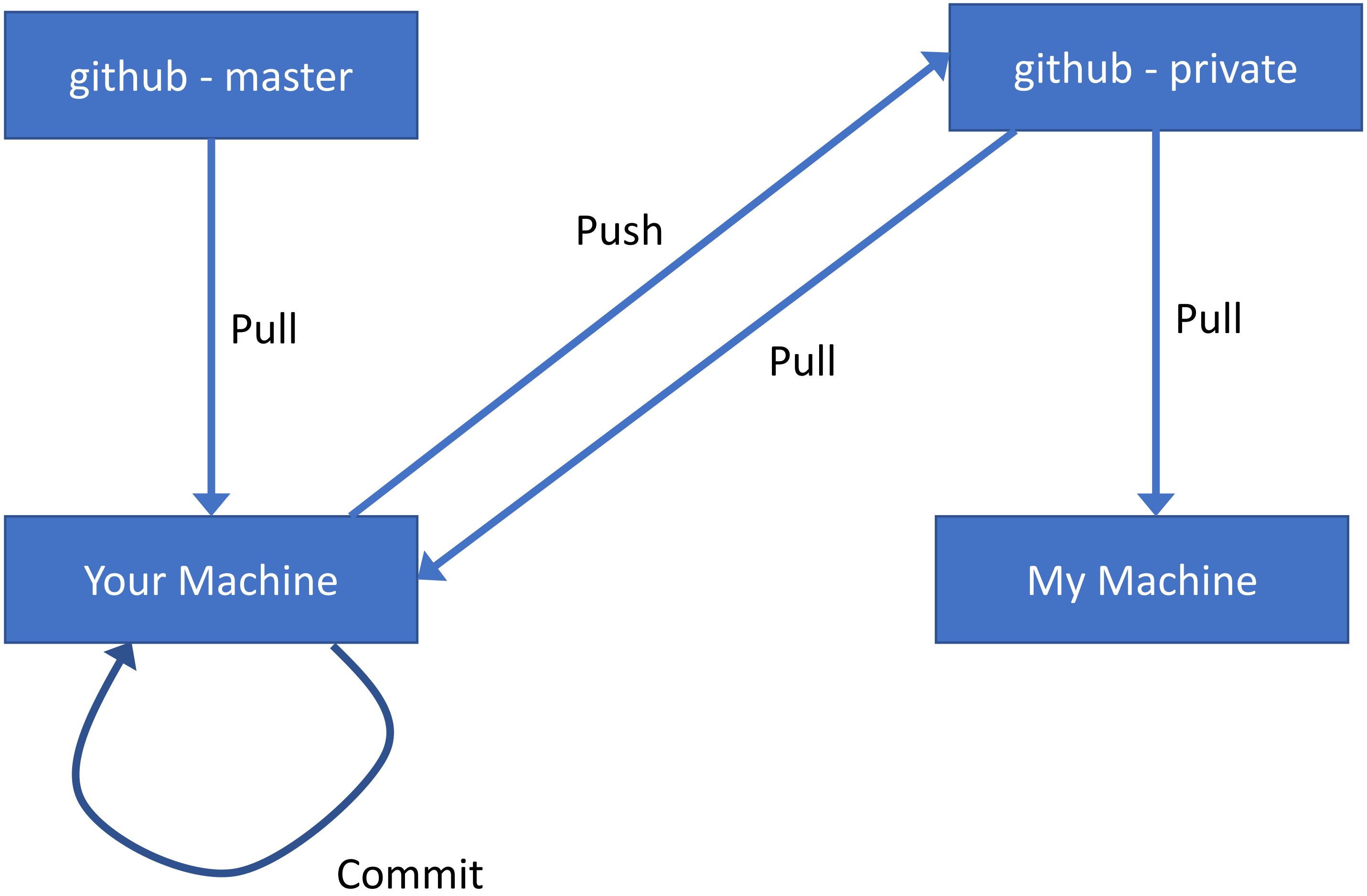
- So far you have used git "linearly"
  - One person (you) working in your own private repo
  - Each commit follows sequentially from the previous one
- Except... there were two people in each repo
  - You (via private) and me (via master)
  - Private and master are copies of the same repo
  - We were collaborating on a single deliverable



Capabilities of git we've seen

- Each commit is a snapshot of your repository
  - It is an exact replica of your source-code at that point
  - It only tracks the files you tell it to
- "Go back to" earlier commits: `git checkout`
- Send your changes to a remote: `git push`
- Integrate changes from a remote: `git pull`
- See the complete list of changes: `git log`

Single user model



github - master

github - private

Push

Pull

Pull

Pull

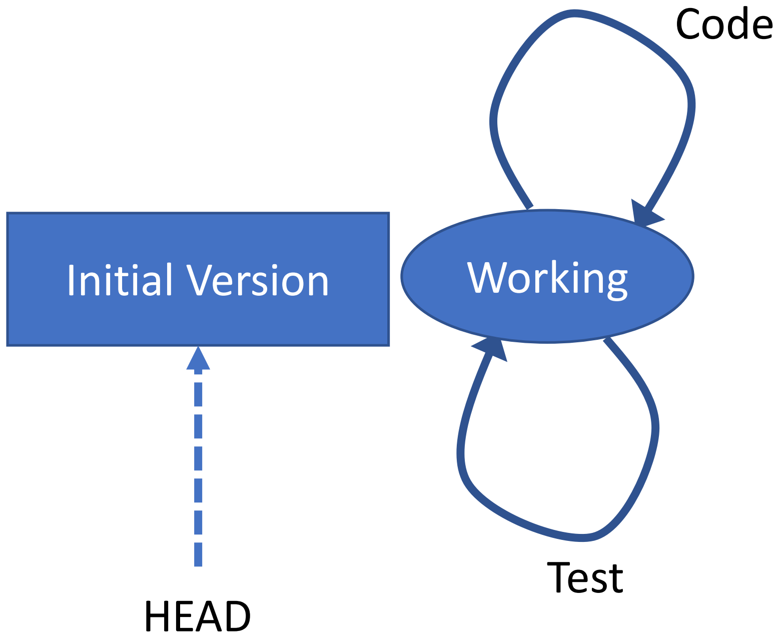
Your Machine

My Machine

Commit

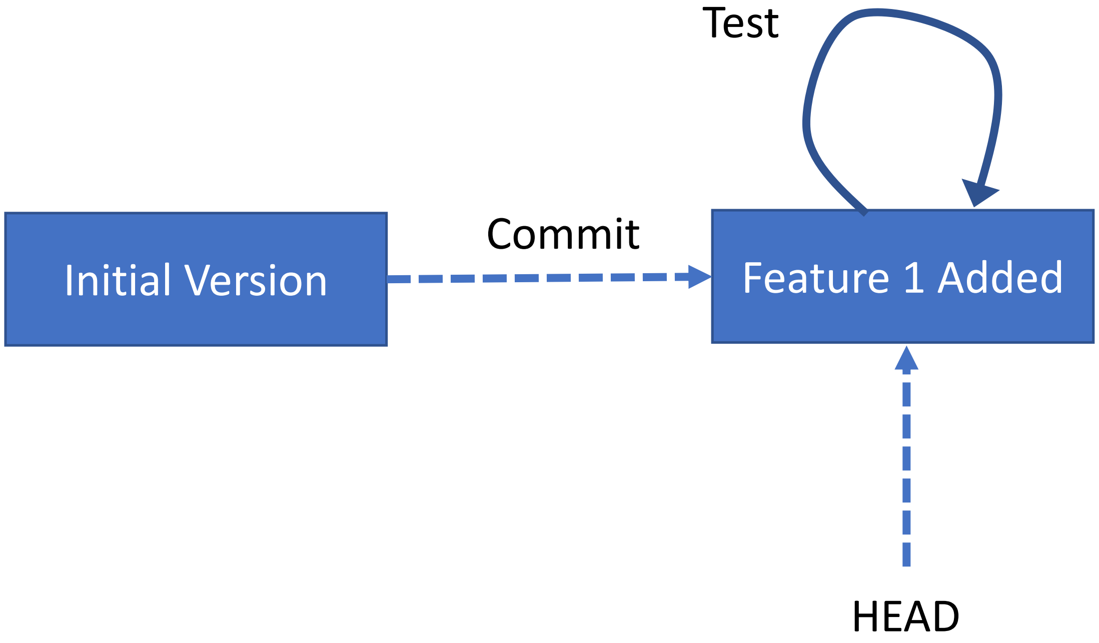
Commits as a sequence of states

- Ideally each commit "works"
  - Each commit should compile and pass its own tests
  - Functionality improves with each commit



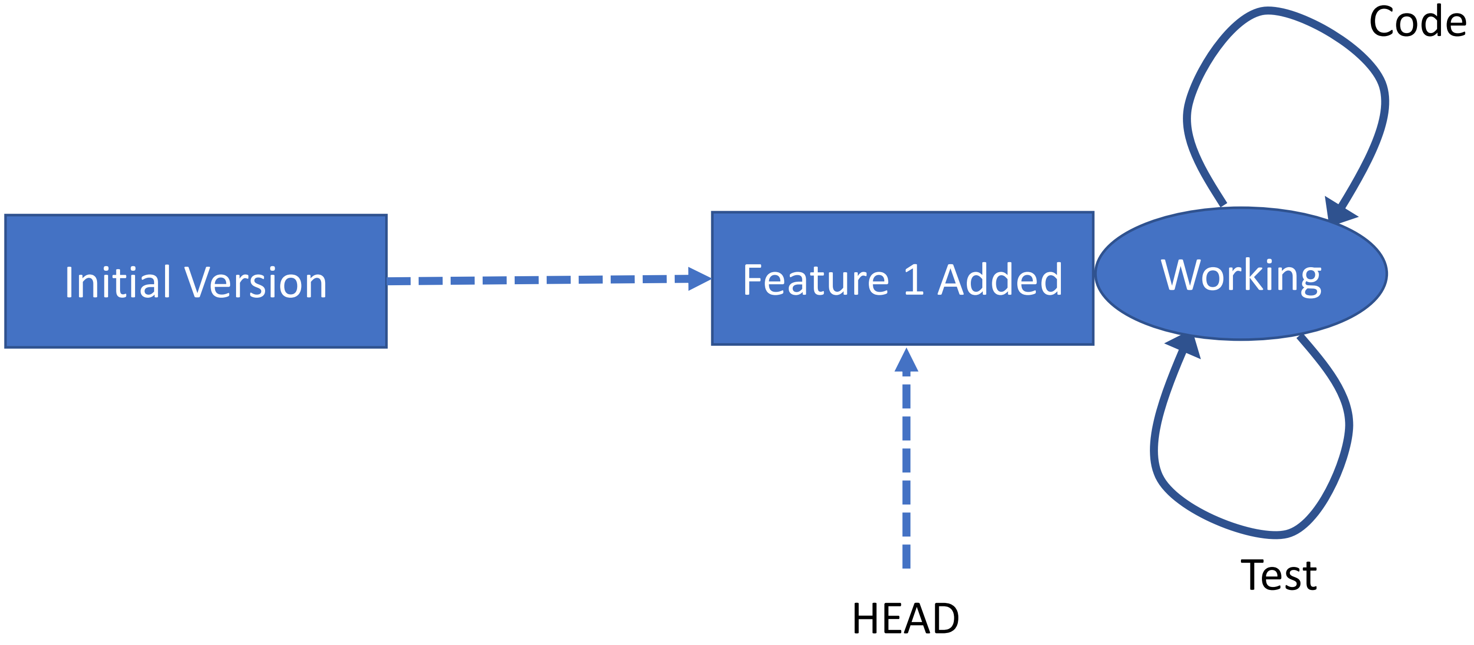
Commits as a sequence of states

- Ideally each commit "works"
  - Each commit should compile and pass its own tests
  - Functionality improves with each commit



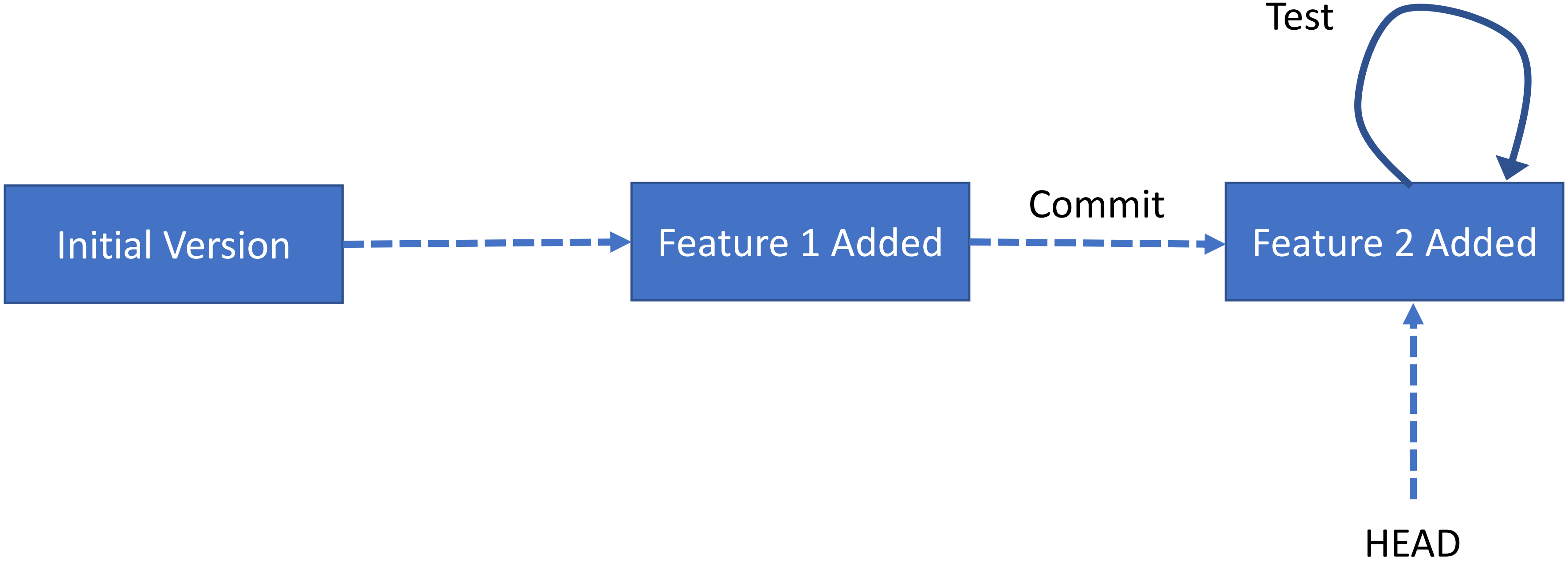
Commits as a sequence of states

- Ideally each commit "works"
  - Each commit should compile and pass its own tests
  - Functionality improves with each commit

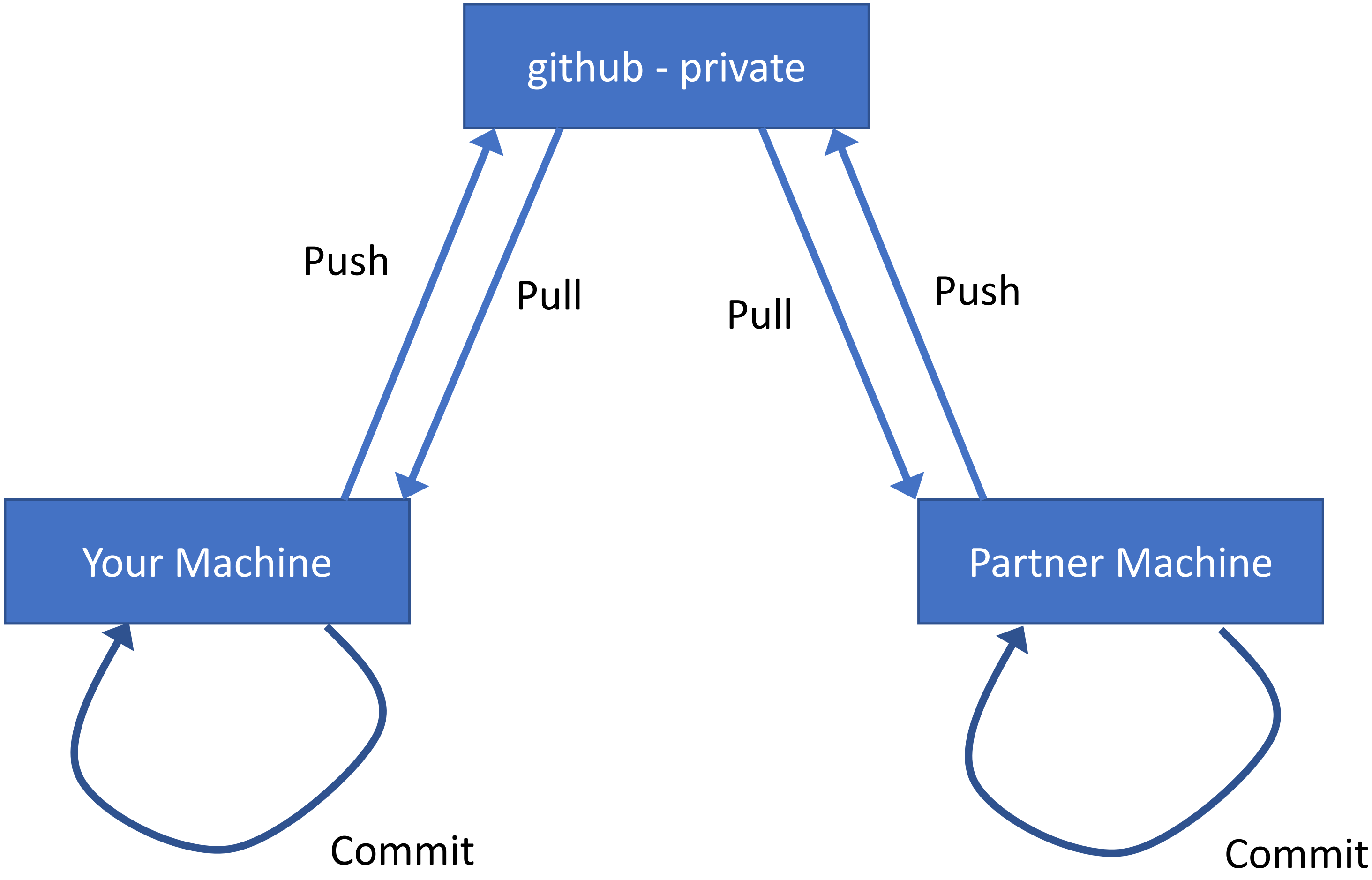


Commits as a sequence of states

- Ideally each commit "works"
  - Each commit should compile and pass its own tests
  - Functionality improves with each commit



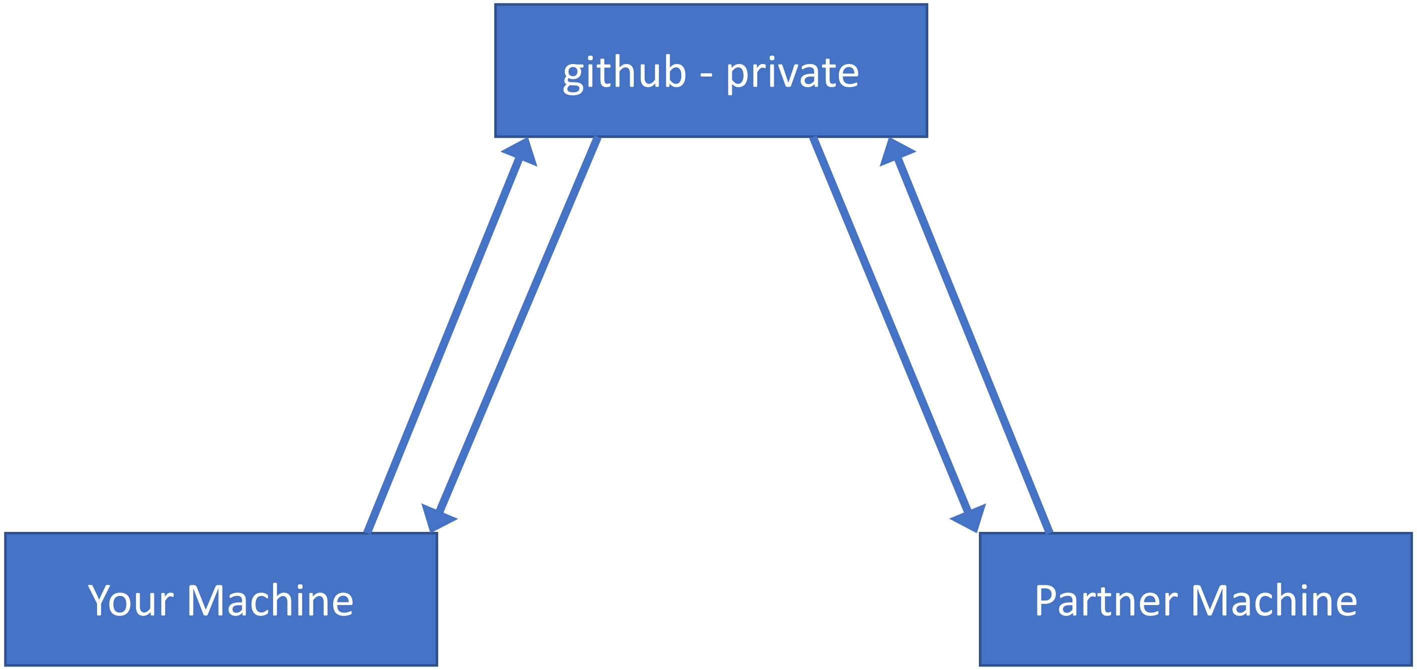
Multi-user model



Conflicts in multi-user repos

- git is designed to support parallel working
  - Each coder works on their own local repo
  - Occasionally you push/pull to synchronise work
- When you "pull" git will look for conflicts
  - A conflict is when two people modify the same thing
- Most of the time conflicts don't occur
  - Changing files in different directories
  - Changing different parts of the same file
- Sometimes you'll change the same line of code
  - git will report a "merge conflict"
  - You (hopefully) have not seen this yet

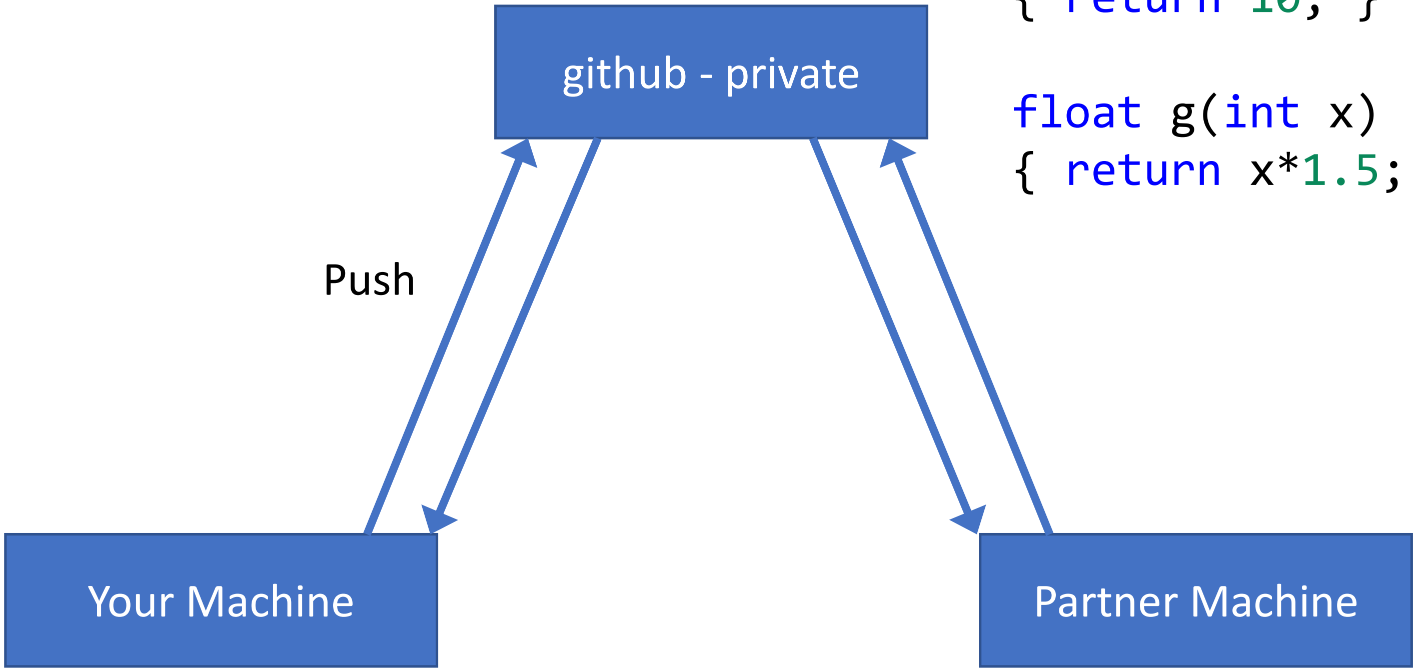
Multi-user model



```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

Multi-user model



```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

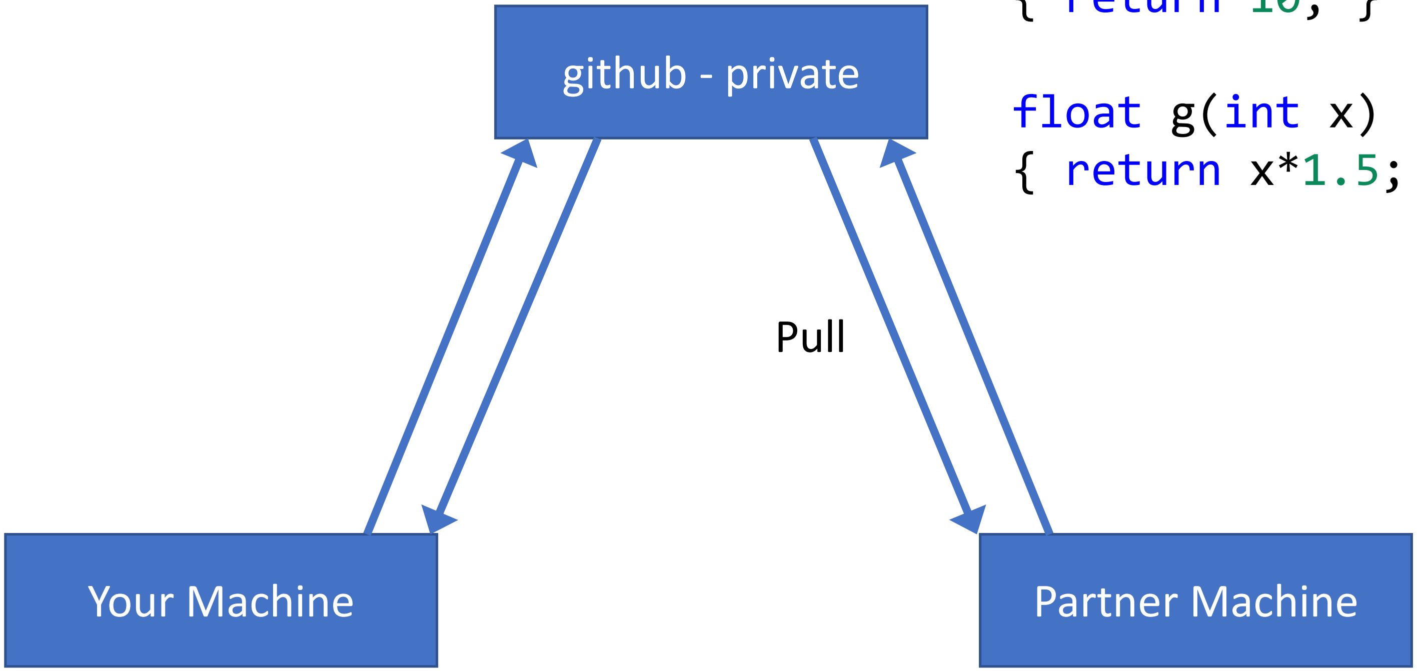
```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

Multi-user model

```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```



```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

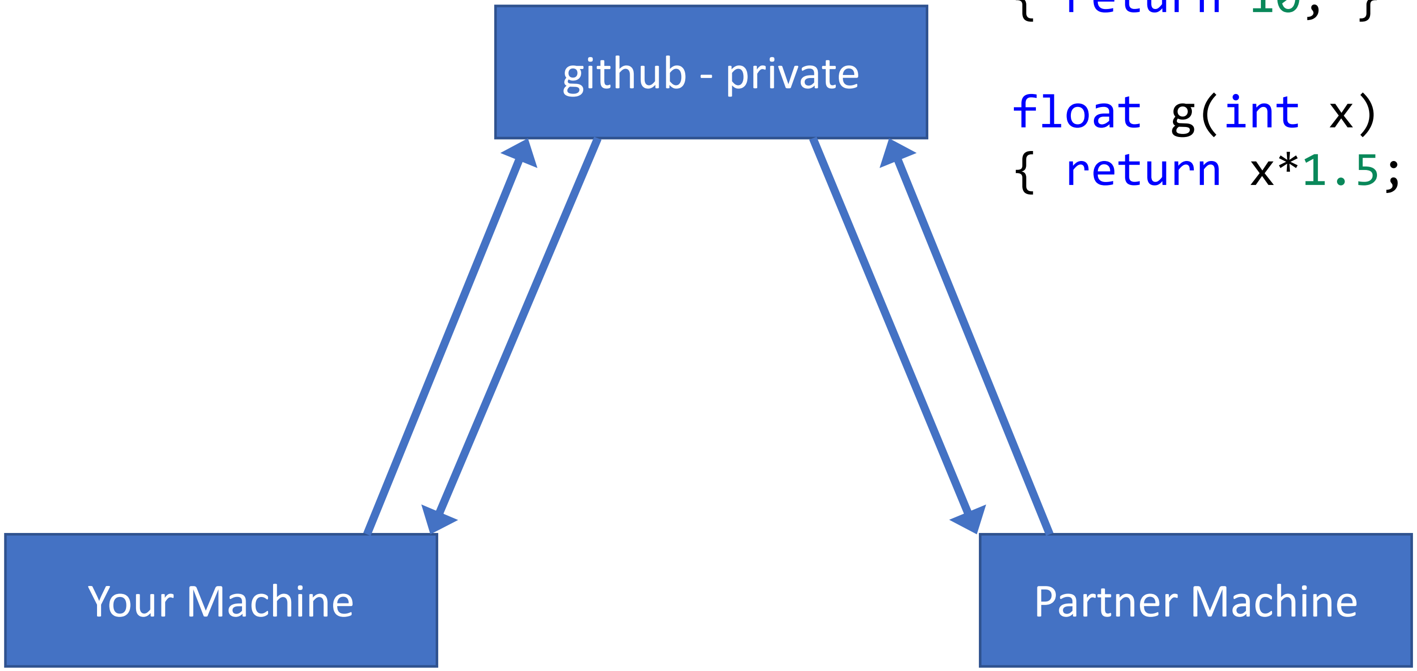
```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

Multi-user model

```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```



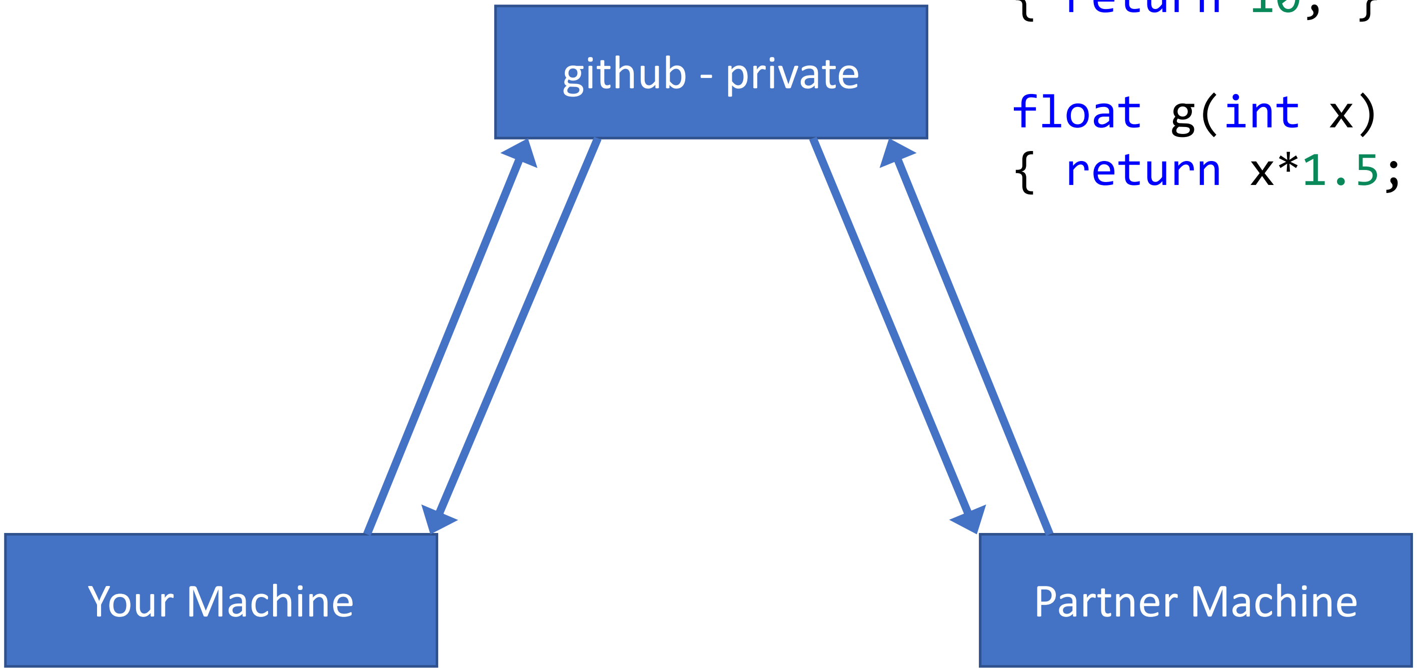
```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

Multi-user model



```
int f()
{ return 10; }

float g(int x)
{ return x*1.5; }
```

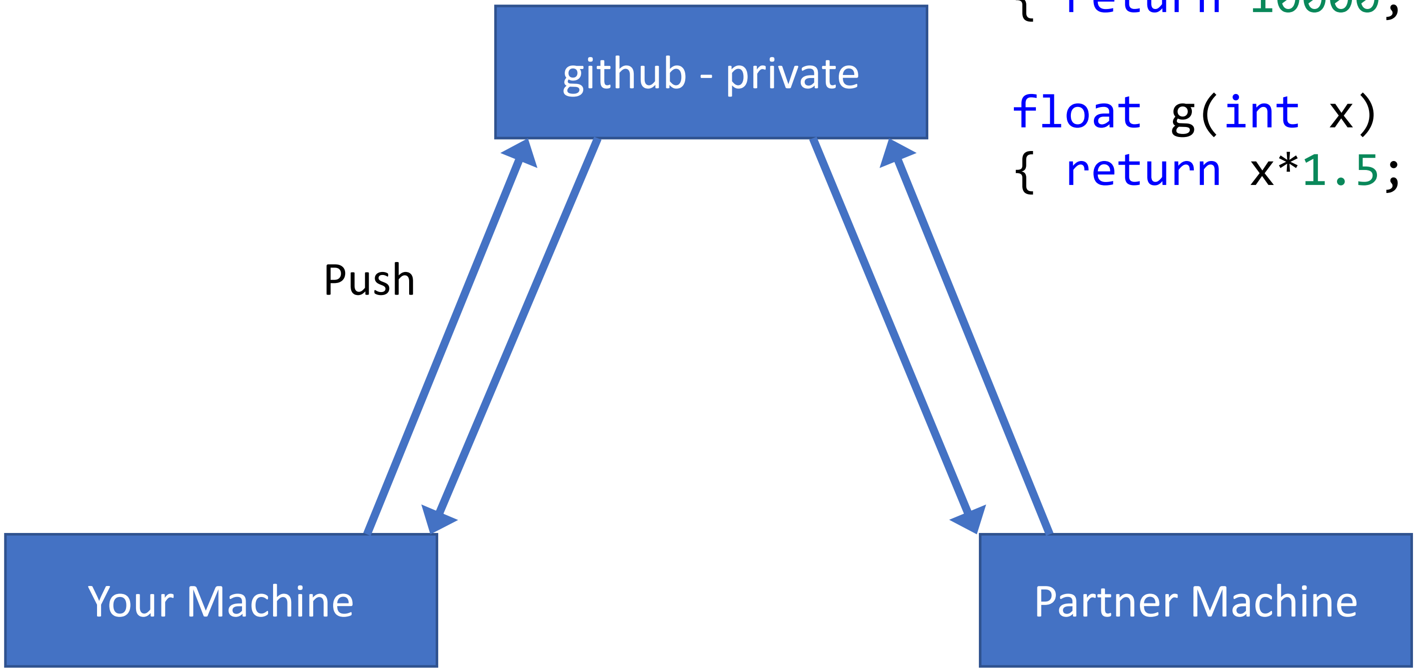
```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```

```
int f()
{ return 10; }

float g(int x)
{ return x*7.555; }
```

Multi-user model



```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```

```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```

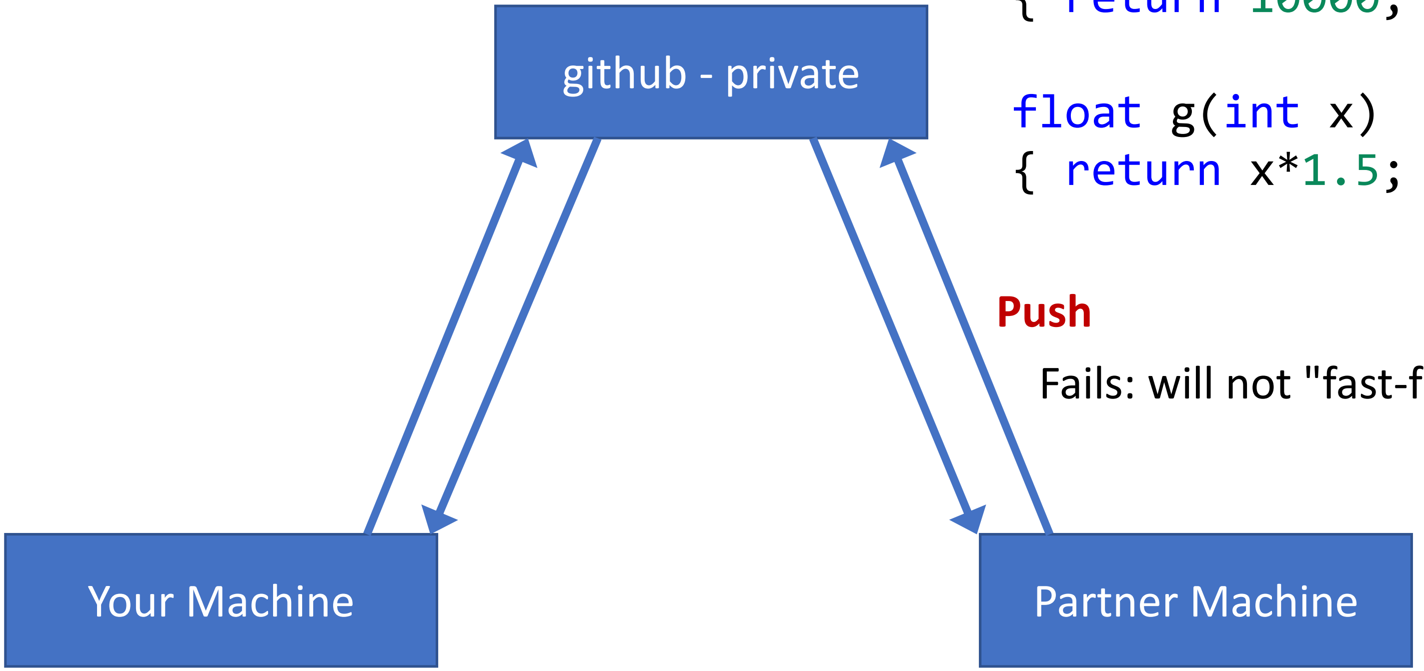
```
int f()
{ return 10; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```



github - private

Your Machine

Partner Machine

Push

Fails: will not "fast-forward"

```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```

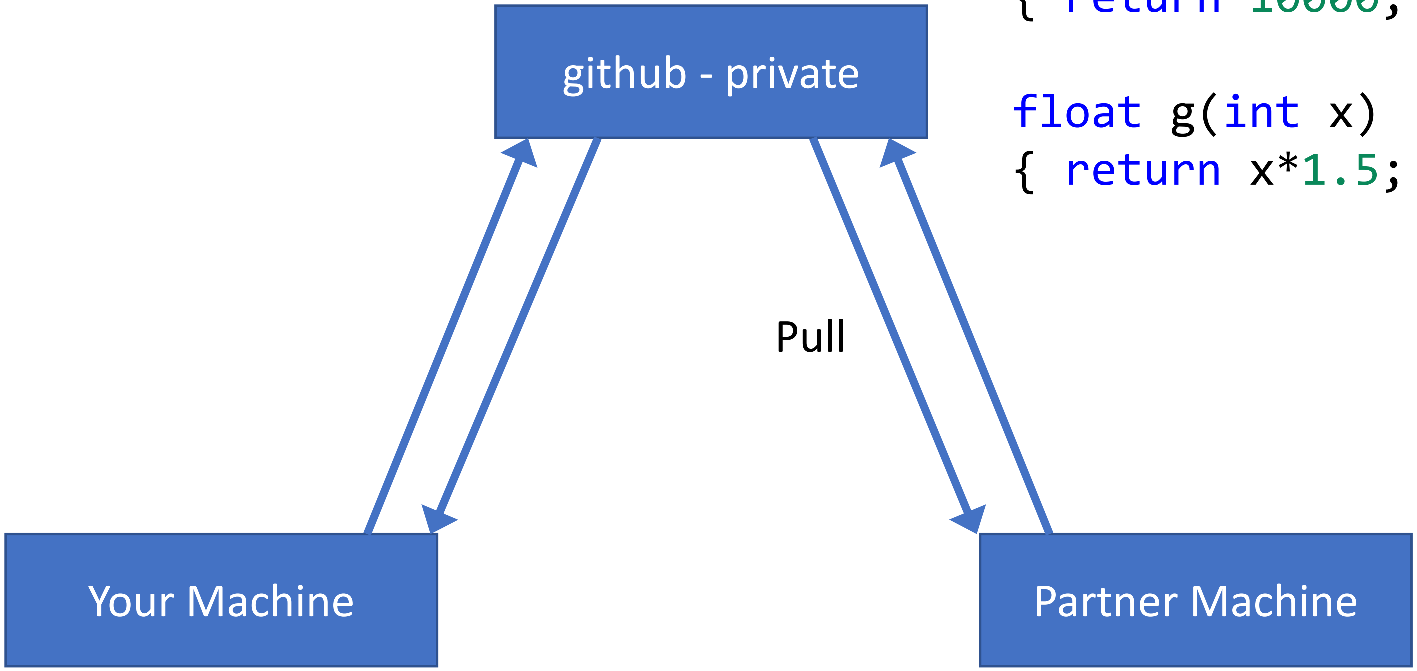
```
int f()
{ return 10; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```



github - private

Pull

Your Machine

Partner Machine

```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```

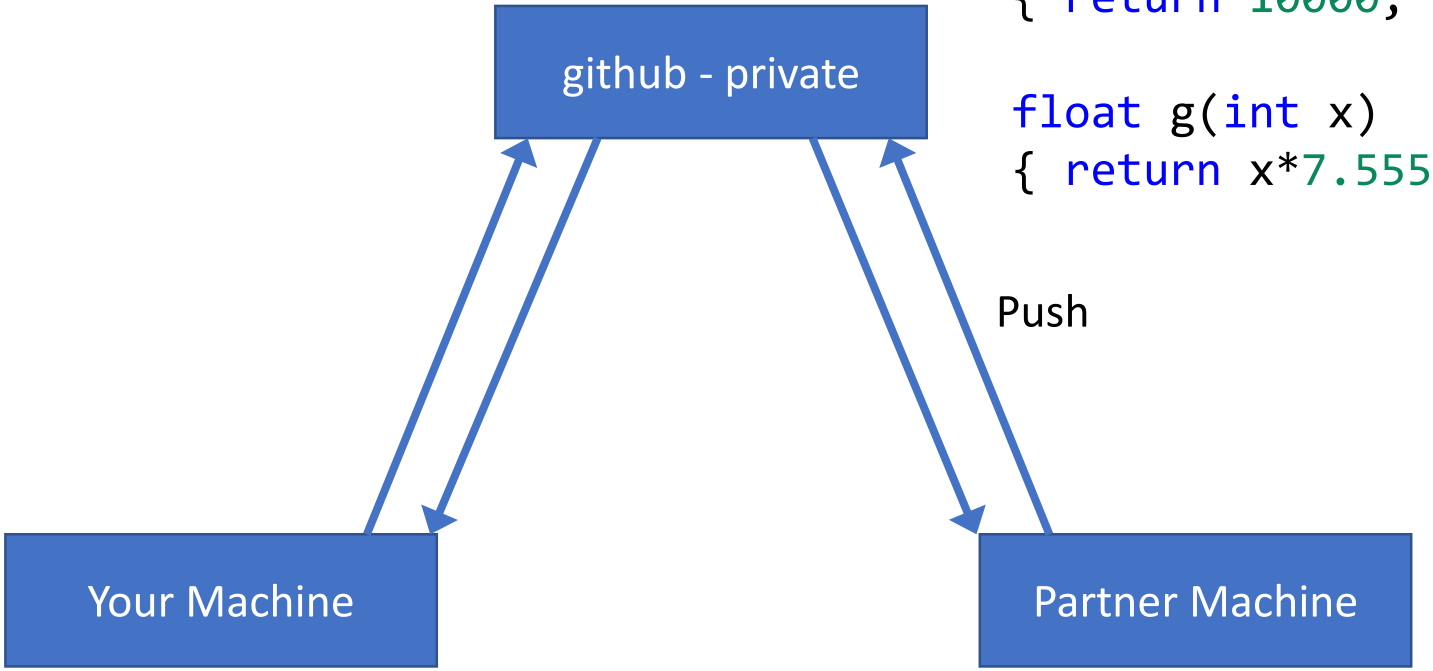
```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```



```
int f()
{ return 10000; }

float g(int x)
{ return x*1.5; }
```

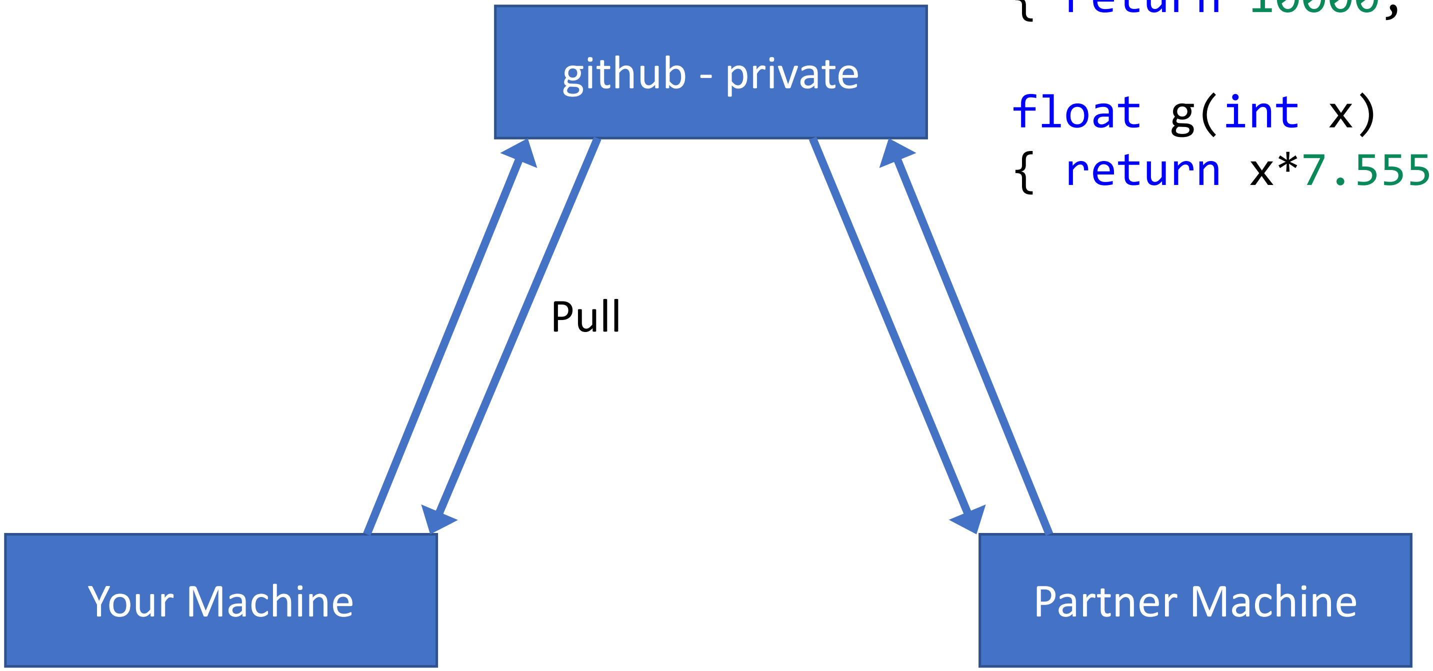
```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```



```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```

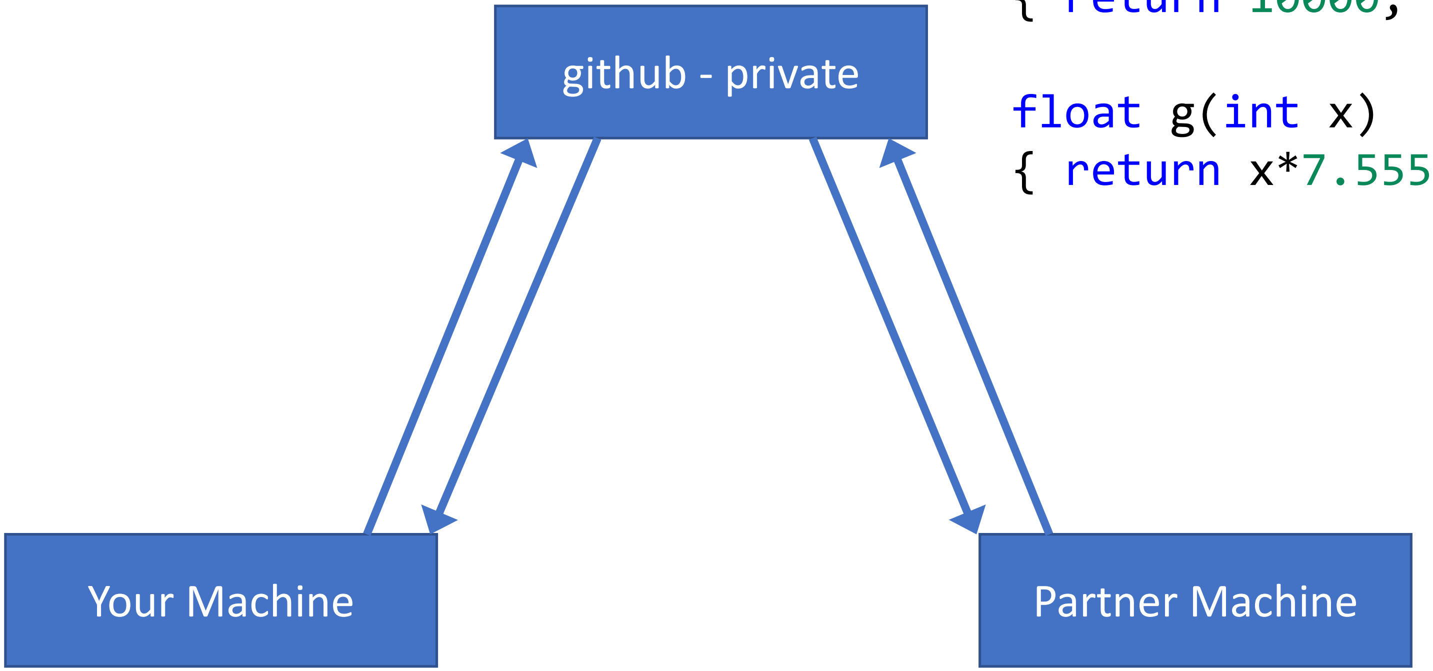
```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 10000; }

float g(int x)
{ return x*7.555; }
```



```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```

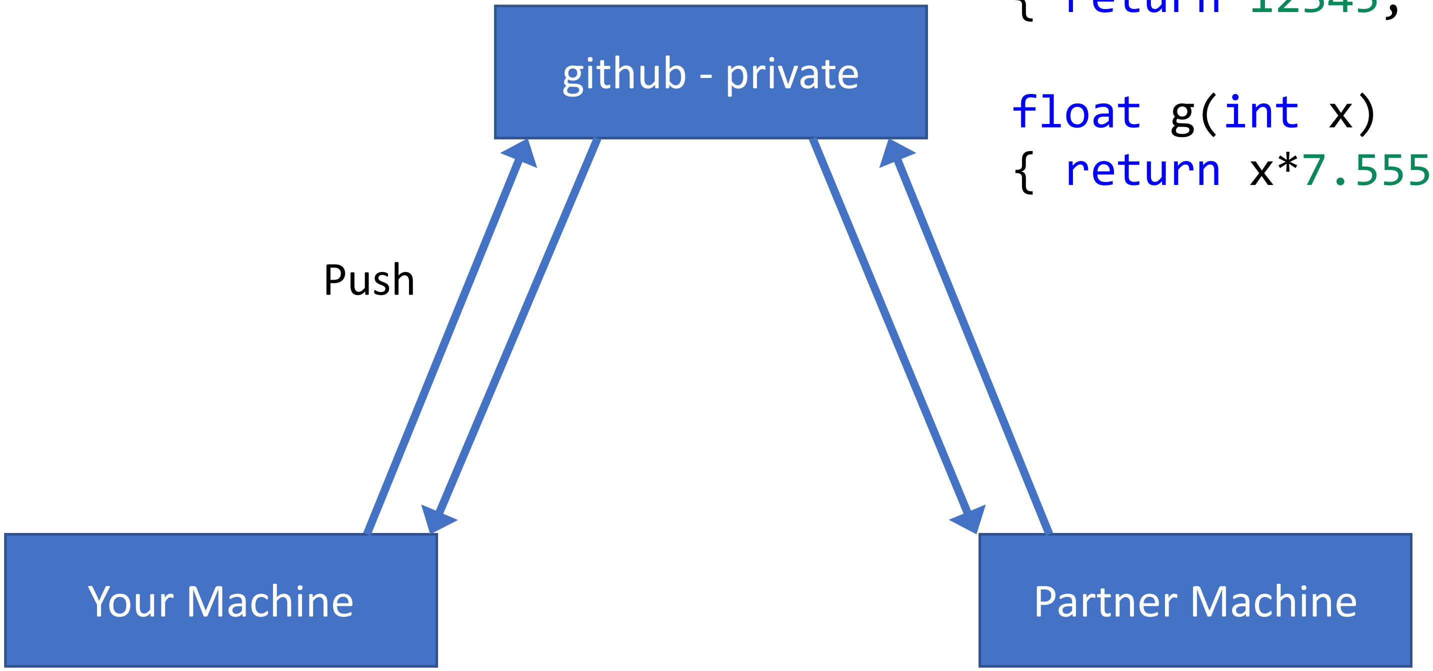
```
int f()
{ return 78901; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```



```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```

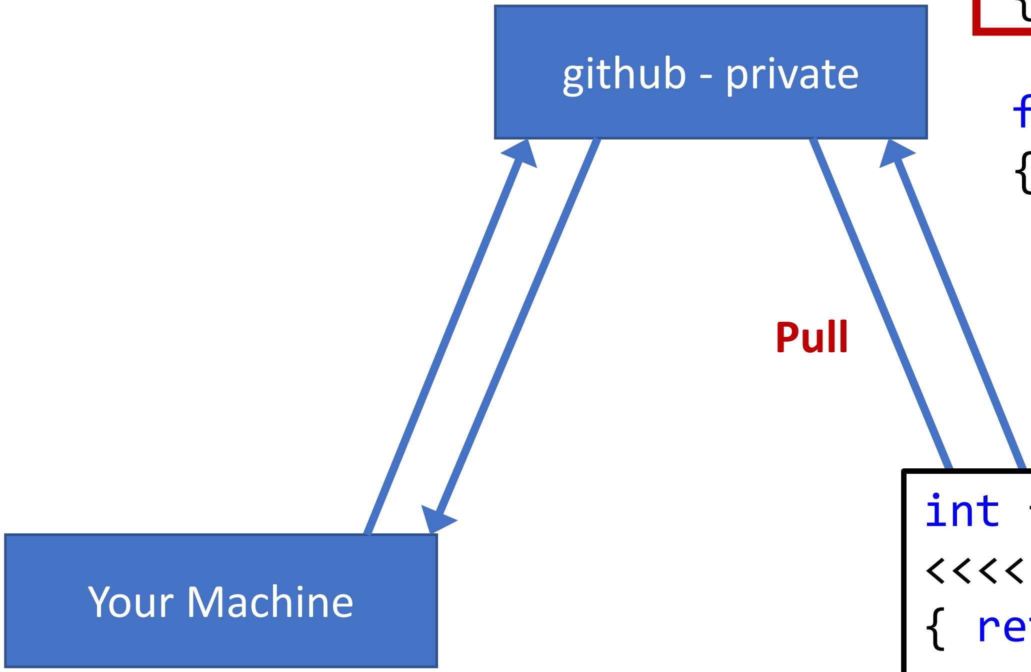
```
int f()
{ return 78901; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```



You must decide how to resolve the conflict

```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```

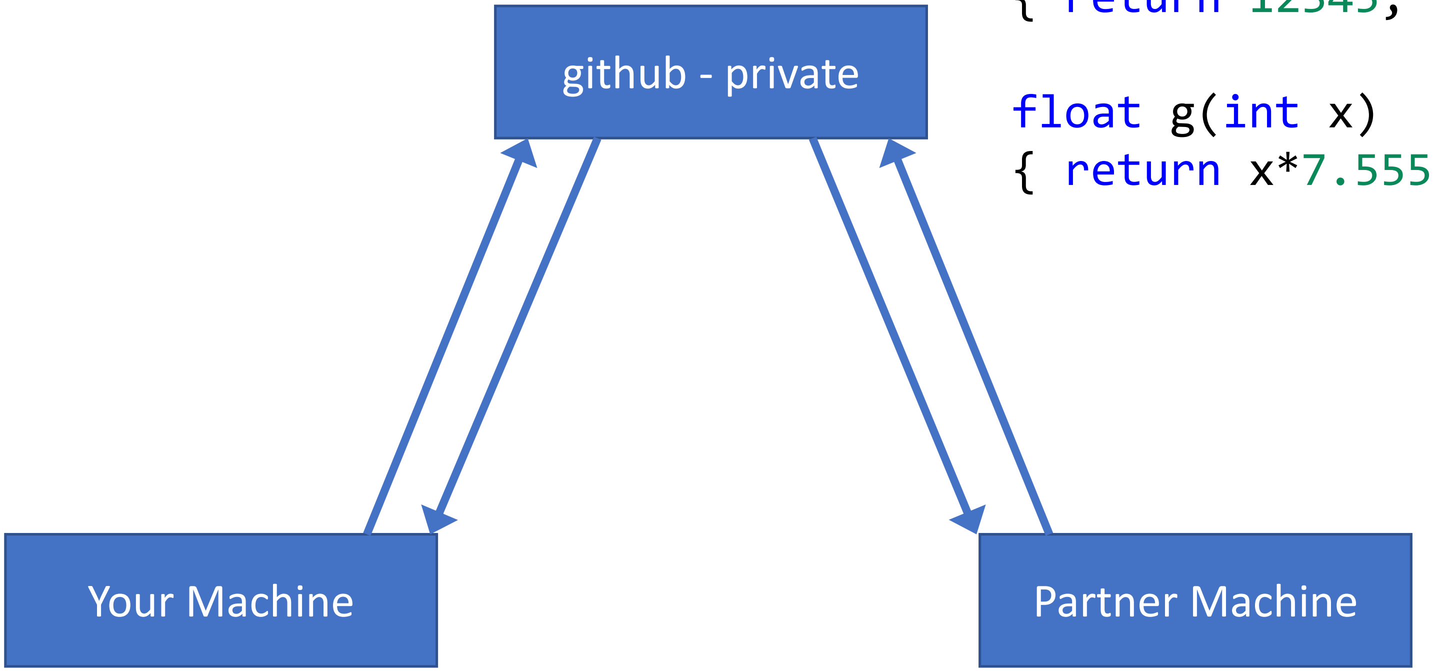
```
int f()
<<<<<< HEAD
{ return 78901; }
======
{ return 12345; }
>>>>>> origin/master

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```



```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```

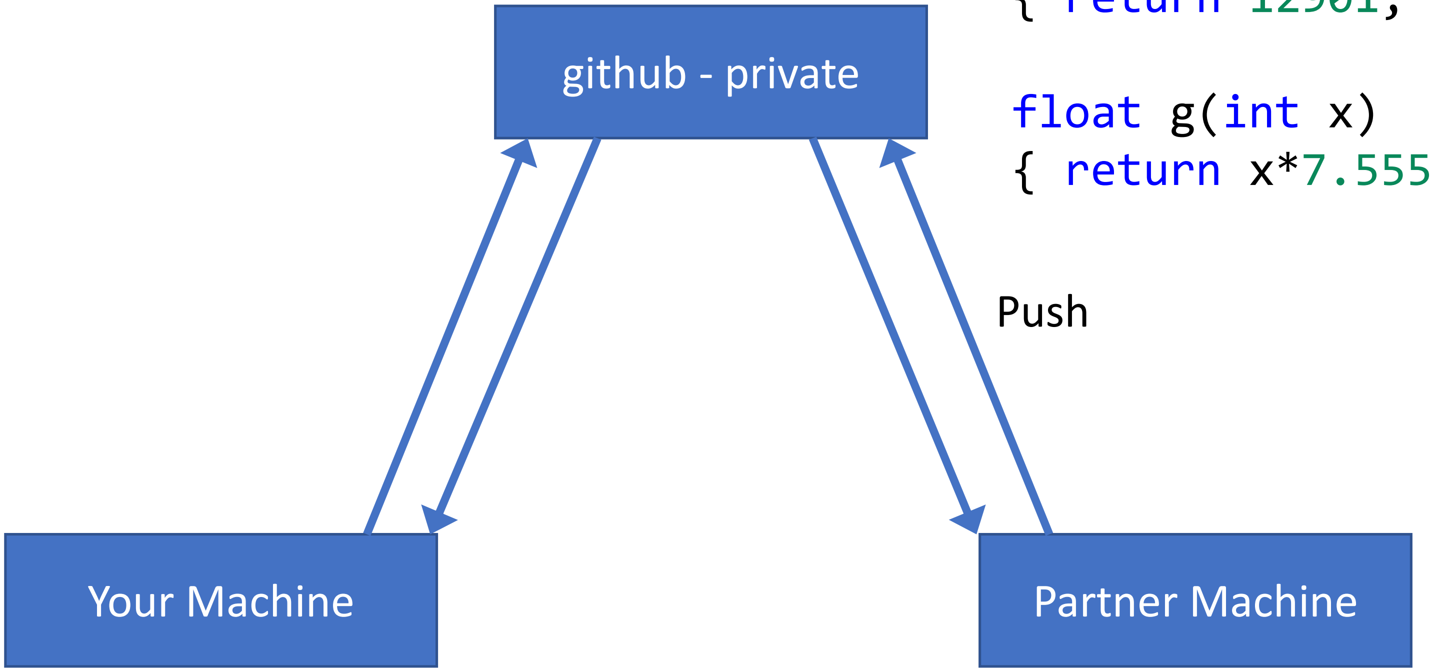
```
int f()
{ return 12901; }

float g(int x)
{ return x*7.555; }
```

Multi-user model

```
int f()
{ return 12901; }

float g(int x)
{ return x*7.555; }
```



```
int f()
{ return 12345; }

float g(int x)
{ return x*7.555; }
```

```
int f()
{ return 12901; }

float g(int x)
{ return x*7.555; }
```

Dealing with merge conflicts

- Occasional merge conflicts are inevitable
  - Try to minimise the number of times they occur
  - Try to minimise their size (lines of code)
- General principles for avoiding conflicts
  - Talk to each other!
  - push and pull fairly often
  - Try to only push code that compiles
  - Try not to work on the same function at the same time

Working together

- This quarter's lab and portfolio is pair work
  - This does not have to be the final pair
- You'll set up your pairs in the github repos yourself
  - Takes effect immediately (no manual step from me)
  - Instructions are in the lab
  - Pairs will be extracted from the repo information
  - A match-making service is available if necessary
- Labs and portfolio are preparing for the "exam"
  - Similar to the activities you'll be asked to do
  - Tasks are also structured and specified in a similar way
  - But: with more discussion, hints, suggestions

Solving a problem

Our goal : implement "git"

- General questions to ask ourselves
  - What is the system supposed to ***do***?
  - Where does the input come from?
  - Where does output go to?
  - What state needs to be maintained?
  - What are the main operations that change state?
  - How can we split the system up into chunks?
  - Can we reduce the scope of the system?

What does git do?

- log – list the known commits
- commit – capture the state of files
- checkout – restore the state of files
- ~~push/pull~~ – *consider out of scope*

What data does git manage?

- There is a thing called a repository
  - A repository has a list of commits
- There is a thing called a commit
  - Commits have:
    - A message
    - A timestamp
    - A hash
    - A parent/previous commit
    - A collection of files
    - The current contents of each file

What is a reasonable API?

```
struct FileInfo
{
    string name;
    string contents;
};

struct Commit
{
    string name;
    timestamp date;
    string hash;
    vector<FileInfo> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

```
struct FileInfo
{
    string name;
    vector<char> contents;
};

struct Commit
{
    string name;
    timestamp date;
    string hash;
    vector<FileInfo> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

What is a reasonable API?

```
struct FileInfo
{
    string name;
    string contents;
};

struct Commit
{
    string name;
    timestamp date;
    string hash;
    vector<FileInfo> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

```
struct Commit
{
    string name;
    timestamp date;
    string hash;
    vector<pair<string,string>> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

What is a reasonable API?

```
struct FileInfo
{
    string name;
    string contents;
};

struct Commit
{
    string name;
    timestamp date;
    string hash;
    vector<FileInfo> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

```
struct Commit
{
    string name;
    timestamp date;
    string hash;
    map<string,string> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

What is a reasonable API?

```
struct FileInfo
{
    string name;
    string contents;
};

struct Commit
{
    string name;
    timestamp date;
    string hash;
    vector<FileInfo> files;
};

struct Repository
{
    vector<Commit> commits;
};
```

```
class Commit
{
public:
  string get_name() const;
  timestamp get_date() const;
  string get_hash() const;
  void checkout() const;
};

class Repository
{
public:
  int count() const;
  const Commit &at(int index) const;
  const Commit &at(string hash) const;
};
```

Choosing a style

Why use objects?

- Objects are there to provide interfaces
  - You cannot understand an entire code-base
  - You might be able to understand object interactions
- Bugs are usually down to corrupted state
  - Calling a function with an invalid parameter
  - Modifying a structure so that it has an invalid state
- *Encapsulation* provides abstracted states via functions
  - Users cannot access member data directly
  - Methods move objects between valid states
  - Users are not able to change or corrupt state

Objects and Polymorphism

“Polymorphism” : *using a single interface (API)*
to access multiple types or implementations

We have seen three types of polymorphism:

- **Overloading** : “ad-hoc” polymorphism
- **Templates**: parametric polymorphism
- **Inheritance** : sub-type polymorphism

Ad-hoc polymorphism: *overloads*

- Define different functionality for a fixed set of types
  - We have to manually provide a definition for each type
  - The types must be known at compile-time
  - The function will be selected at compile-time

An example: overloading << for ostream

  - Every class can exploit the interface
  - You have to explicitly provide a new overload

Parametric polymorphism: *templates*

- Define functionality based on an unknown type T
  - Only one definition of function or class is needed
    - We don’t know what the type T is when writing code
  - The types must be known at compile-time
  - The compiler will specialise code at compile-time

- Many examples in the STL:
  - Container classes : `vector<T>, list<T>`
  - Algorithms and functions: `min<T>, sort<T>`

Sub-type polymorphism: *inheritance*

- Extend and refine functionality of a base type
  - The base class defines a general class of behaviour
  - Any number of classes can derived from the base
  - The choice of implementation is made at run-time

- We’ve seen examples of this:
  - *Shapes*: dynamic selection of draw() or area()
  - *Rover*: dynamic selection between svg and action output

How to choose?

- No-one can tell you when to use each approach
  - Inheritance is not always the answer
  - Objects are not always the answer
- Good interfaces comes from experience
  - Trying to design interfaces (and realising why they are bad)
  - Reading documentation for existing interfaces
  - Extending existing interfaces and libraries
- APIs are harder and more valuable than code
  - Anyone can implement a function
  - Designing a system is much harder

Convergent vs divergent solutions

- Academic assessment tends to be "convergent"
  - There is a well-specified problem
  - It has one exact solution solution
- Engineering practice is "divergent"
  - There is a loosely-specified problem
  - There is an infinite space of possible solutions
  - It is sometimes difficult to say whether solution is "right"
- Here we are aiming for a mixture
  - There is a well-specified problem
  - It has an infinite space of possible solutions
  - There are well-defined ways of saying if it is right
  - Beyond that: *we do not care how the problem is solved*