Code Review

SWPP

April 16th

Juneyoung Lee

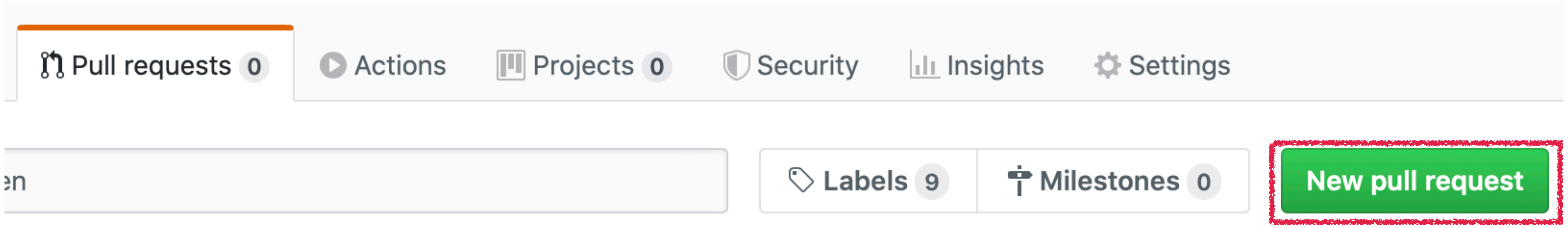
Assignment 5. Code Review + GoogleTest

- We’ll announce the assignment later (maybe next week)

- We’ll give code to each other & have a code review
  - 2~3 per a student
- You will write simple unit tests using GoogleTest framework
  - LLVM 10.0 already has GoogleTest framework, so nothing to prepare in advance

Working as a Team

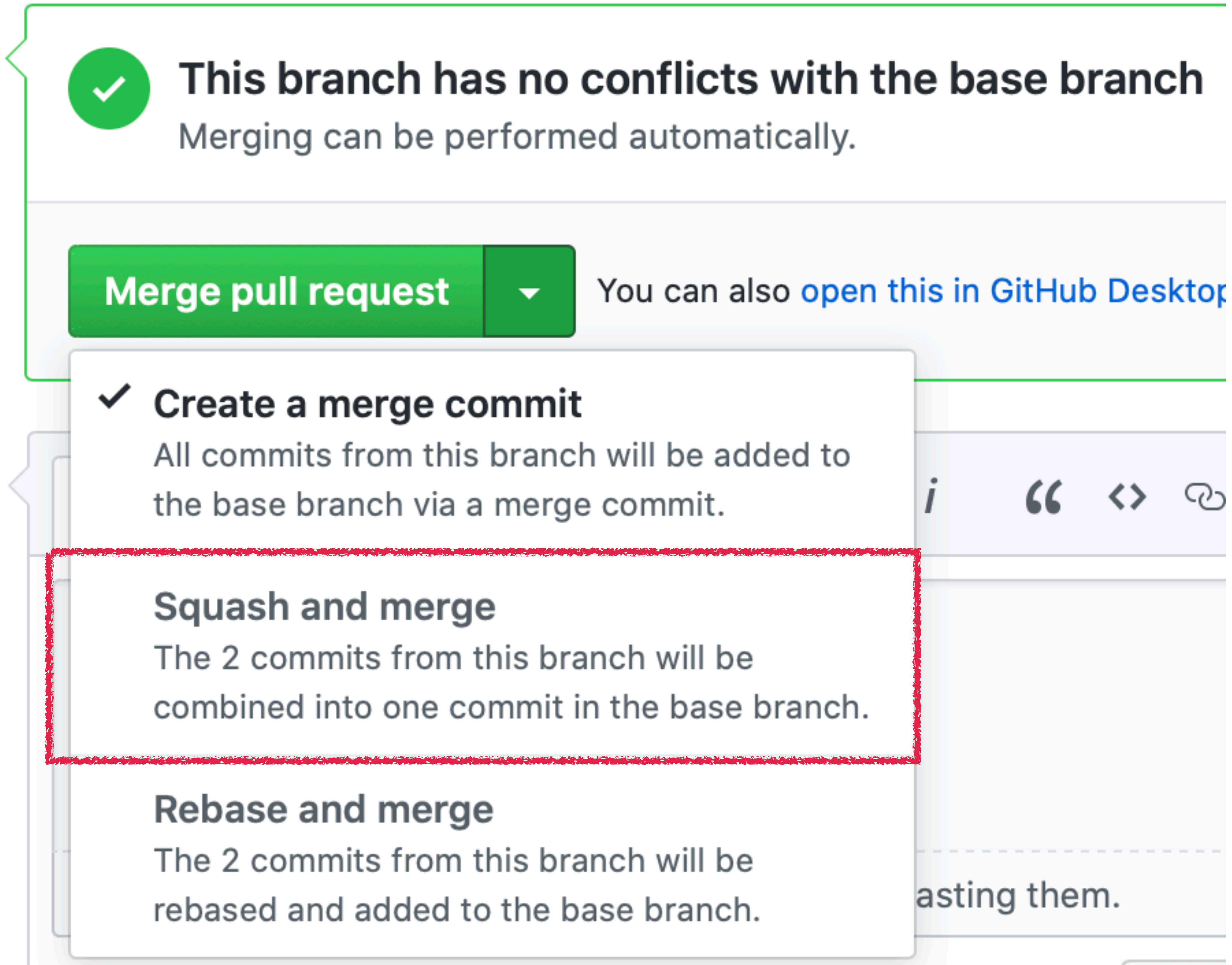
- In the project, you’re going to work as a team
- Each team will have one main repository
- Each teammate will fork the main repository & work on it
- After you finish your implementation, you can send a pull request to the main repo



- After the pull request is accepted by reviewers, it is merged into main repo’s master

Pull Request

- A pull request is a unit of working feature
  - After it is merged, it should be compiled & all existing unit tests should pass
- Making git history linear is important for finding out a buggy pull request
- Using ‘squash and merge’ will help this



This branch has no conflicts with the base branch

Merging can be performed automatically.

You can also open this in GitHub Desktop

✓ **Create a merge commit**

All commits from this branch will be added to the base branch via a merge commit.

Squash and merge

The 2 commits from this branch will be combined into one commit in the base branch.

Rebase and merge

The 2 commits from this branch will be rebased and added to the base branch.

asting them.

Before Sending Pull Request..

- Before *starting* implementation: please share what you're going to do with people
  - This will help reduce burden of reviewers
  - Will prevent collision
- If you're going to implement a complex algorithm..
  - Having short slides for reviewers will be great (imagine you are a reviewer!)
  - This will help finding potential bugs in advance as well

Code Review Process

- Understanding the submitted patch & giving feedbacks (or accepting with 'LGTM')
- It is often slow & take longer time than writing a code alone
- It takes your emotional energy as well (Imagine someone attacked your code)
- But in the long run, the program becomes less buggy & simpler, so it takes less time
- Code review is essential when you are writing a very complex program

Reviewer's Work

- Reviewers can do 5 things:
  1. Check whether the PR correctly addresses the issue
  2. Point out possible correctness/performance issues (e.g. UB, value copy)
  3. Check whether existing libraries can be used (e.g. std::sort())
  4. See whether better C++ idiom can be used (e.g. using template)
  5. Check whether there is a missing test (e.g. FileCheck, GoogleTest)

1. Check whether the PR correctly addresses the issue

- Reviewee should write the description of PR succinctly
  - Imagine you are a reviewer! :)
- Reviewers can request reviewee to explain it via Skype/Phone call
- Reviewers can ask ‘why implement this way?’ and ‘why not implement this way?’
- Actually, this pass alone can find numerous bugs from the patch

2. Point out possible correctness/performance issues

- Most common cases in C++ (in my opinion):

A. Out-of-bounds access

B. Use-after-free

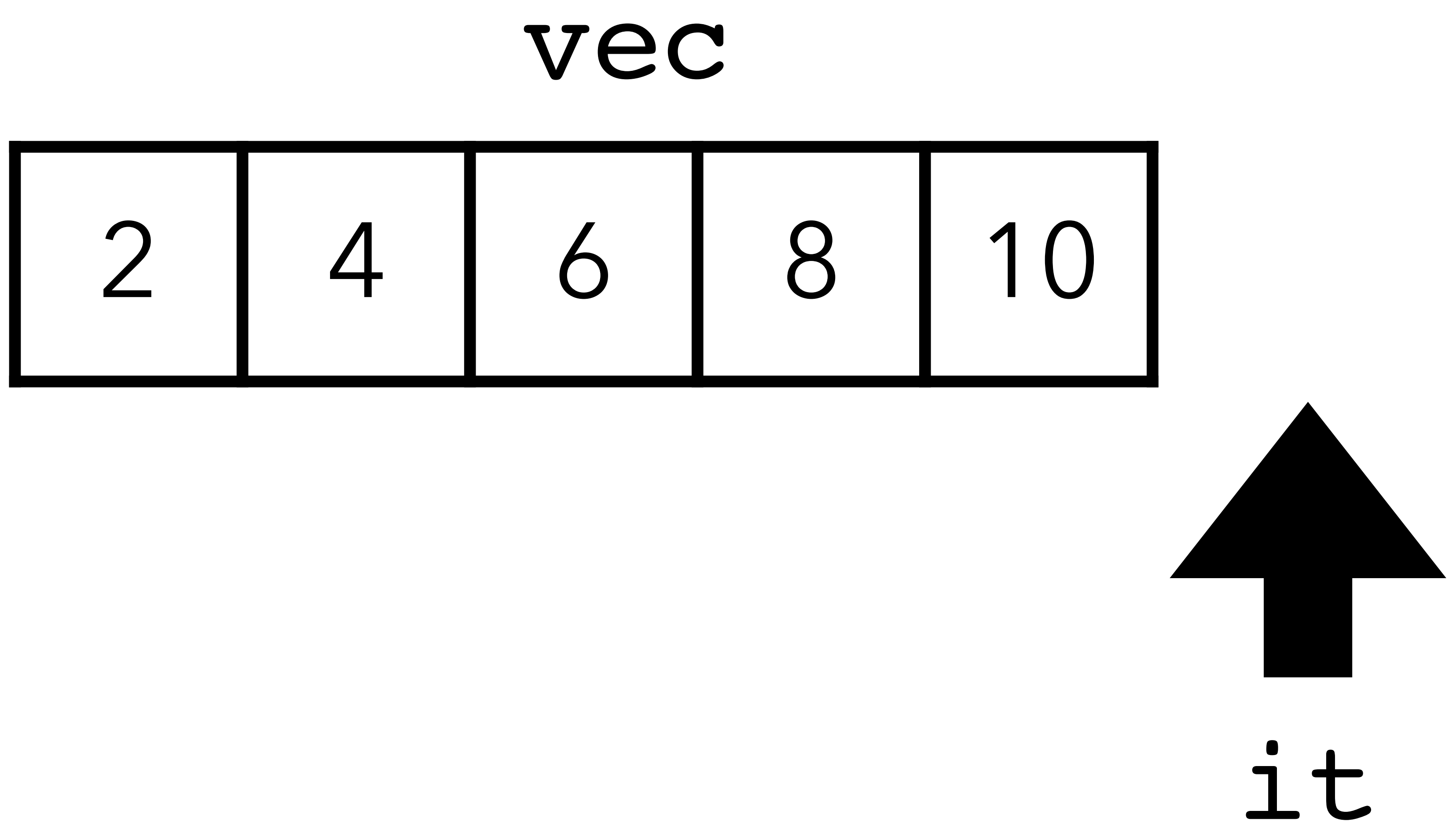
C. Passing a container by value

D. Integer overflows

A. Out-of-bounds access

```
auto it = std::find(vec.begin(), vec.end(), 5);
*it = 6;
```

```
auto it = std::find(vec.begin(), vec.end(), 5);
if (it != vec.end())
  *it = 6;
```



B. Use-after-free

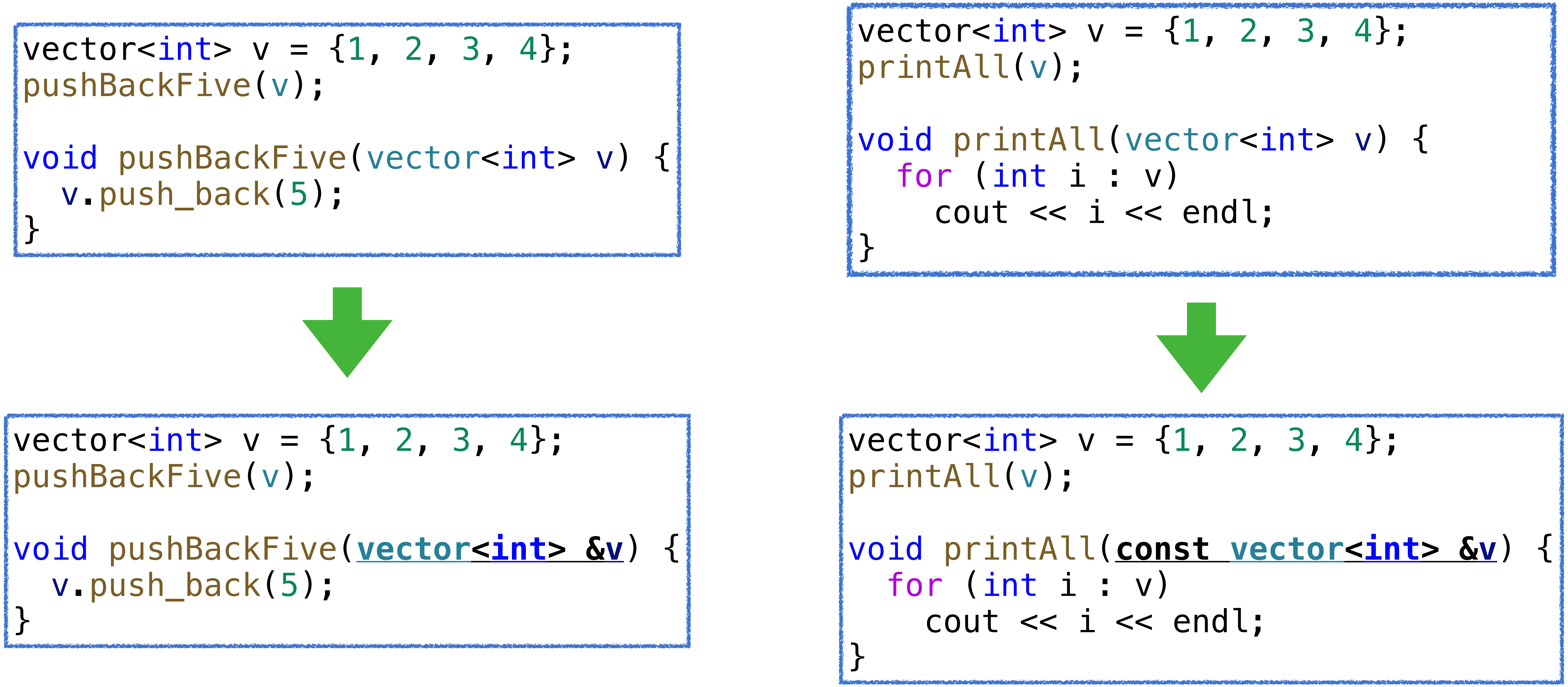
```
auto it = std::find(vec.begin(), vec.end(), 5);
if (it != vec.end())
  vec.erase(it);

cout << “after 5: “;
for (; it != vec.end(); ++it)
  cout << *it << “ ”;
```

```
auto it = std::find(vec.begin(), vec.end(), 5);
if (it != vec.end())
  it = vec.erase(it);

cout << “after 5: “;
for (; it != vec.end(); ++it)
  cout << *it << “ ”;
```

C. Passing a container by value



```
vector<int> v = {1, 2, 3, 4};
pushBackFive(v);

void pushBackFive(vector<int> v) {
  v.push_back(5);
}
```

```
vector<int> v = {1, 2, 3, 4};
pushBackFive(v);

void pushBackFive(vector<int> &v) {
  v.push_back(5);
}
```

```
vector<int> v = {1, 2, 3, 4};
printAll(v);

void printAll(vector<int> v) {
  for (int i : v)
    cout << i << endl;
}
```

```
vector<int> v = {1, 2, 3, 4};
printAll(v);

void printAll(const vector<int> &v) {
  for (int i : v)
    cout << i << endl;
}
```

D. Integer Overflows

```
vector<int> v = {1, 2, 3, 4};
for (unsigned i = v.size() - 1; i >= 0; --i)
  if (v[i] % 2 == 0)
    even_count++;
```

```
vector<int> v = {1, 2, 3, 4};
for (unsigned i = 0; i < v.size(); ++i)
  if (v[v.size() - i - 1] % 2 == 0)
    even_count++;

// Or, use int, or rbegin() + rend()
```

How to Catch These Bugs?

- They are syntactic patterns; review process can effectively catch these bugs
- You can use clang-tidy (https://clang.llvm.org/extra/clang-tidy/checks/bugprone-string-integer-assignment.html).
- Making diff smaller can also help reviewers detect problems.

Assertions

- Writing assertions is a very good practice for detecting a bug in advance!
    - Helps people understand which invariant should be met at the point
    - Helps find bugs by catching an inconsistent state in advance
- C++ idiom: assert(Predicate && “Something is wrong”);
- When compiled with -g option & linked with LLVM, it will show stack trace with line numbers. :)

N Reviewers, 1 Reviewee

- Reviewee should try hard to reduce burden of reviewers
  1. One PR should have one subject
  2. If PR is too big, split it
  3. Minimize diffs
  4. Attach PR with results on datasets (if possible)

1. One PR should have one subject

Common mistakes:

- Add feature A + format code in feature B
- Add feature A + make tweaks in B to make A faster
- Fix a bug in A, B where A and B are irrelevant
- Add a big feature A + enable it by default
  - Ideally, enabling it should be a separate patch, so reverting it can still preserve code

2. If PR is too big, split it

- 200 lines without tests / comments is assumed to be ‘big’
- You may think your PR cannot be splitted, but it is often true that it can be.
- Common cases:
  - Implement A with complex algorithm -> Implement A + add a better algorithm
  - Fix bugs A, A’, A’’ -> Fix A first + expand it to A’ + A’’

3. Minimize diffs

- Very important for reviewers to understand the code
- Whenever possible, try to stay succinct
  - Use for-each statement
  - Auto type to hide unnecessary details
  - Use standard libraries often (e.g. std::copy, std::sort, etc)
  - Remove unnecessary changes in irrelevant code (e.g. space)