Static typing

Catching errors at compile time

Alexander Heinrich

5th August 2021

Static typing

- Goals of this talk
  - Make code more checkable by tools
  - Reduce accidental errors and typos
  - Express assumptions about the code by using the type system
- Non-goals
  - Prevent intentional misuse
  - Formally verifying software and algorithms

Basics

```
function useNumber(int number) {
   ...
}

useNumber("Hello World"); // compile time error
```

Basics

- Types are guarantees about possible values/states
- Enforced at compile time before even running the code
- Examples

```
Bool = True or False
Uint8 = From 0 to 255
String = From "" to "...", including "255"
```

Dynamic typing

```
function useValue(value) {

    ... // is value a number?

}
```

Dynamic typing

```
function useValue(value) {
  if(isNumber(value)) {
    ...
  }
}
```

Dynamic typing

```
function useValue(value) {
  if(isNumber(value)) {
    ... // guaranteed to be a number here
  }
}
```

Dynamic typing

```
function useValue(value) {
  if(isNumber(value)) {
    printNumber(value)
  }
}
```

Dynamic typing

```
function useValue(value) {
  if(isNumber(value)) {
    printNumber(value) // guarantee gets lost
  }
}
```

Dynamic typing

```
function printNumber(value) {
    ...
}

function useValue(value) {
  if(isNumber(value)) {
    printNumber(value) // guarantee gets lost
  }
}
```

Dynamic typing

```
function printNumber(value) {
    ... // is value a number?
}

function useValue(value) {
  if(isNumber(value)) {
    printNumber(value) // guarantee gets lost
  }
}
```

Possible solutions

1. Pretend type errors can't happen
2. Excessive if statements
3. Static typing

Static typing

- Can help making error handling mandatory

Static typing

- Can help making error handling mandatory

```
function useNumber(int number) {
   ...
}
```

Static typing

- Can help making error handling mandatory

```
function useNumber(int number) {
   ...
}

String input = console.readLine();
```

Static typing

- Can help making error handling mandatory

```
function useNumber(int number) {
  ...
}

String input = console.readLine();

  useNumber(input) // compiler error
```

Static typing

- Can help making error handling mandatory

```
function useNumber(int number) {
  ...
}

String input = console.readLine();
if(int number = parseInt(input)) {
  useNumber(number)
}
```

Static typing

- Can help making error handling mandatory

```
function useNumber(int number) {
  ...
}

String input = console.readLine();
if(int number = parseInt(input)) {
  useNumber(number) // guaranteed to be a number here
}
```

Static typing

- Can help making error handling mandatory

```
function useNumber(int number) {
  ...
}

String input = console.readLine();
if(int number = parseInt(input)) {
  useNumber(number) // guaranteed to be a number here
}
```

- `useNumber()` can't be called when conversion fails
- Success is encoded in the type `int`

Static typing

Static typing doesn't help much when overusing the same types for everything

```
class Settings {
  int id;
  int handle;
  int timestamp;
  int permissions;
  int lookup_key;
  ...
}
```

```
class Settings {
  String id;
  String handle;
  String timestamp;
  String permissions;
  String lookup_key;
  ...
}
```

Example

```
String getCityName(int postal_code) {
   ...
}

getCityName(57072) // "Siegen"
```

Example

```
String getCityName(int postal_code) {
   ...
}

getCityName(57072) // "Siegen"
getCityName(04103) // problem: pending zeroes get lost
```

Example

```
String getCityName(int postal_code) {
   ...
}

getCityName(57072) // "Siegen"
getCityName(04103) // problem: pending zeroes get lost
```

Lets try replacing `int` with `String`

Example

```
String getCityName(String postal_code) {
   ...
}

getCityName("57072") // "Siegen"
getCityName("04103") // "Leipzig"
```

Example

```
String getCityName(String postal_code) {
   ...
}

getCityName("57072") // "Siegen"
getCityName("04103") // "Leipzig"
getCityName("Hello World") // Problem
```

Example

```
String getCityName(String postal_code) {
   ...
}

getCityName("57072") // "Siegen"
getCityName("04103") // "Leipzig"
getCityName("Hello World") // Problem
```

How to handle errors?

- Throwing exception → Runtime
- Returning empty string → Runtime

How does static typing help?

- It can't prevent all runtime errors because invalid values occur only at runtime
- But it allows encoding successful checks/conversions in a type

Example

```
class PostalCode {



  constructor(String code);

  String code;
};
```

Example

```
class PostalCode {



private:
  constructor(String code); // can't be called directly

  String code;
};
```

Example

```
class PostalCode {

  static Optional<PostalCode> parseCode(String code);

private:
  constructor(String code); // can't be called directly

  String code;
};
```

Example

```
class PostalCode {
  // only way to construct object
  static Optional<PostalCode> parseCode(String code);

private:
  constructor(String code); // can't be called directly

  String code;
};
```

Example

```
function sendPackage(PostalCode code) {
  ...
}


  sendPackage("Test 123") // won't compile
```

Example

```
function sendPackage(PostalCode code) {
  ...
}

if(PostalCode code = parseCode("Test 123")) {
  sendPackage(code)
}
```

Example

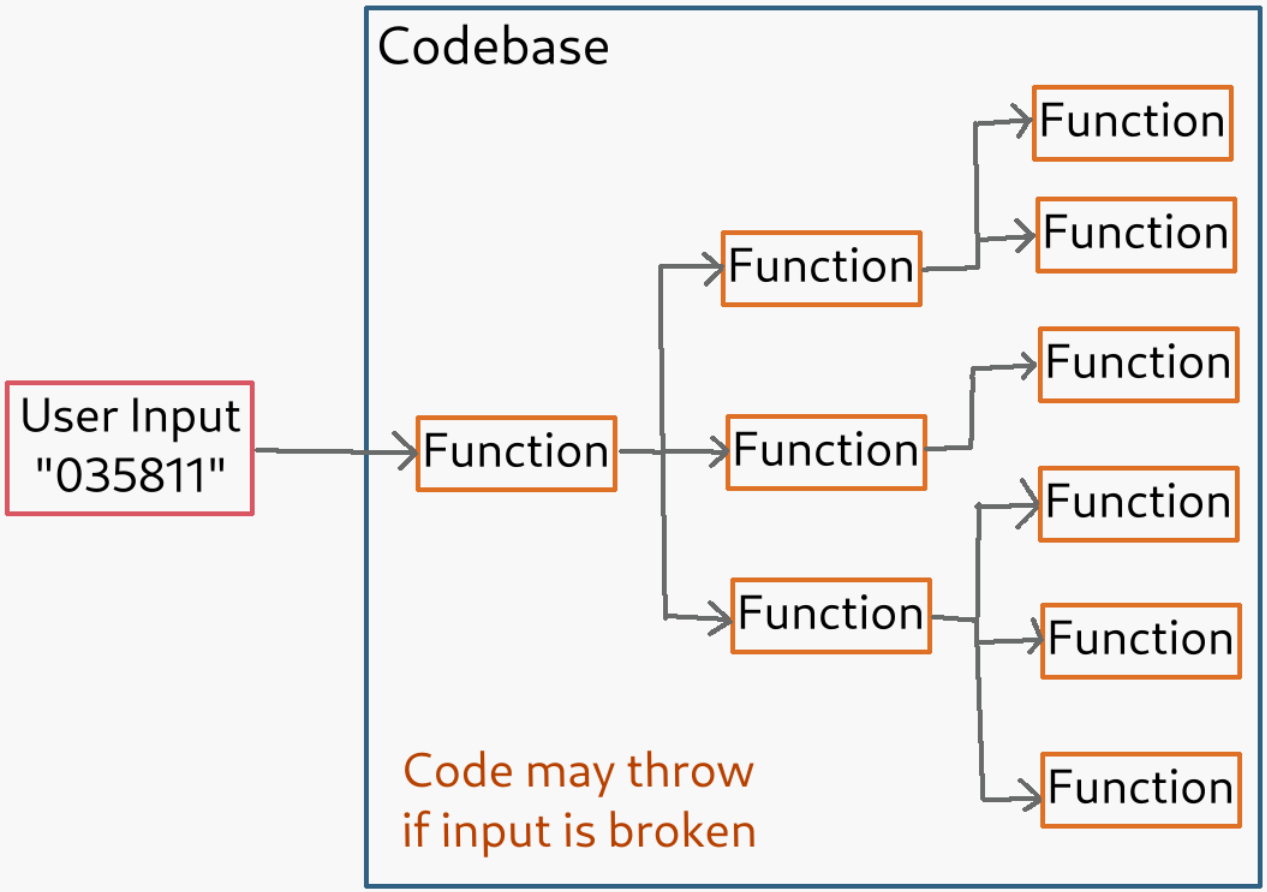
```
function sendPackage(PostalCode code) {
  ...
}

if(PostalCode code = parseCode("Test 123")) {
  sendPackage(code) // only callable on success
}
```

Example

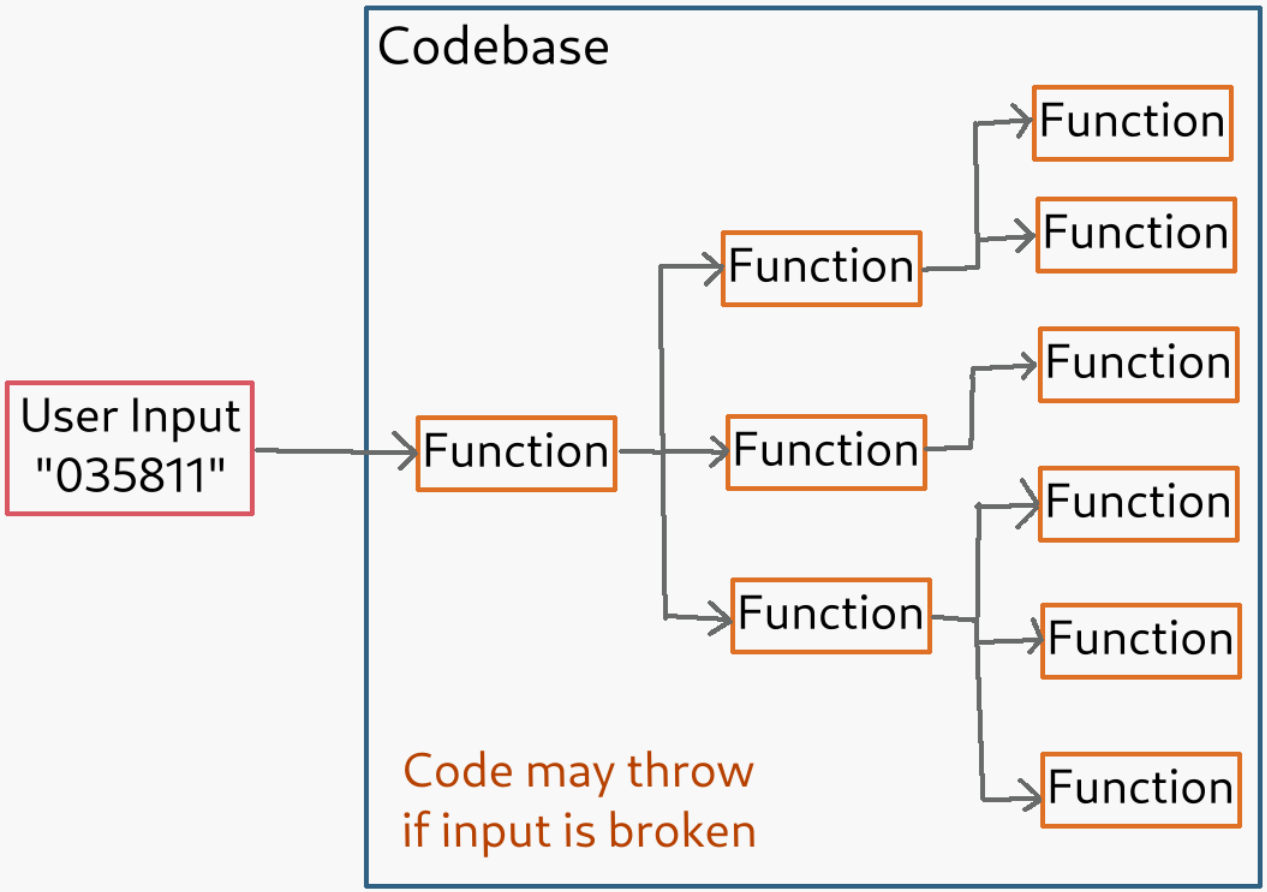
```
function sendPackage(PostalCode code) {
  ...
}

if(PostalCode code = parseCode("Test 123")) {
  sendPackage(code) // only callable on success
}
```

- Responsibility and checks moved from function to its caller

Example

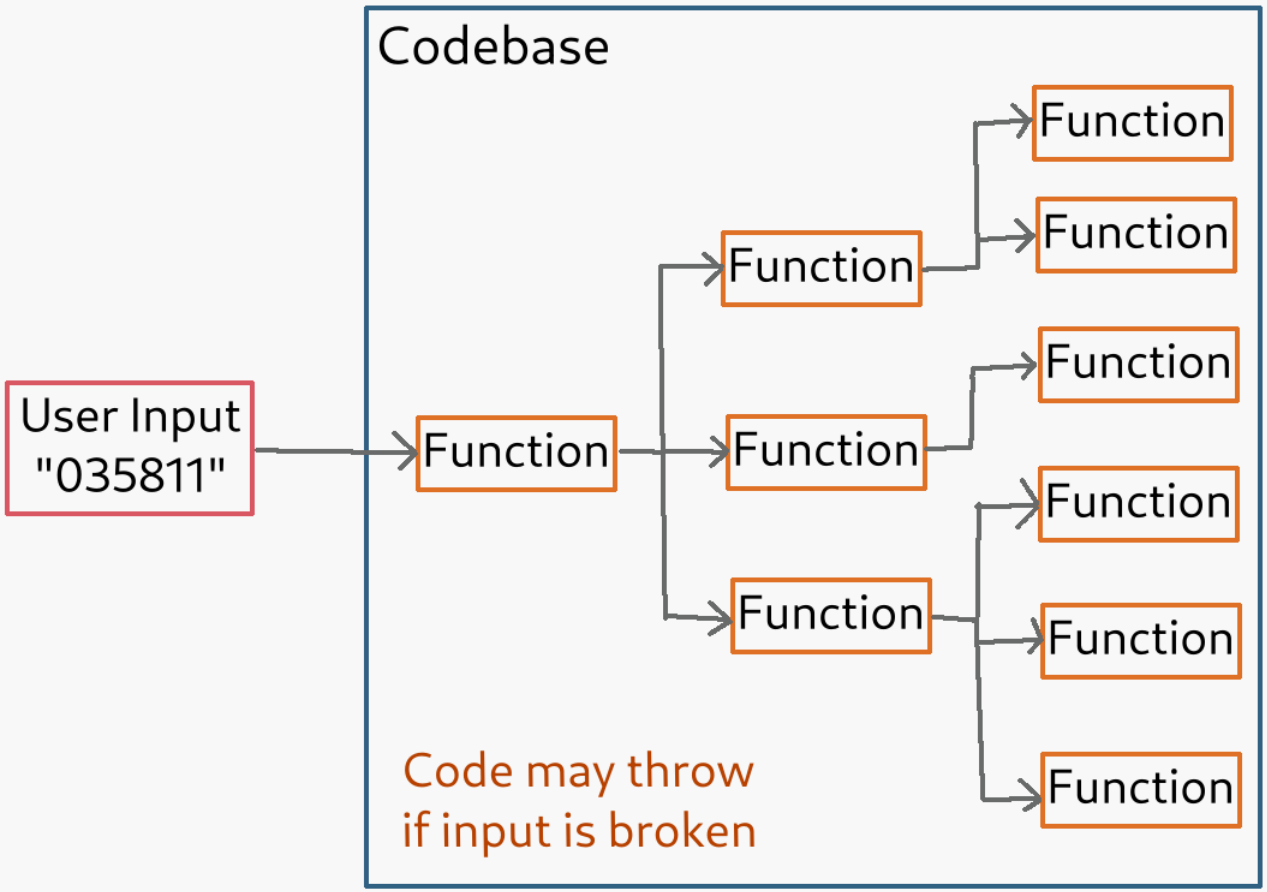

Codebase
User Input
"035811"
Function
Function
Function
Function
Function
Function
Function
Function
Function
Function
Code may throw
if input is broken

Example



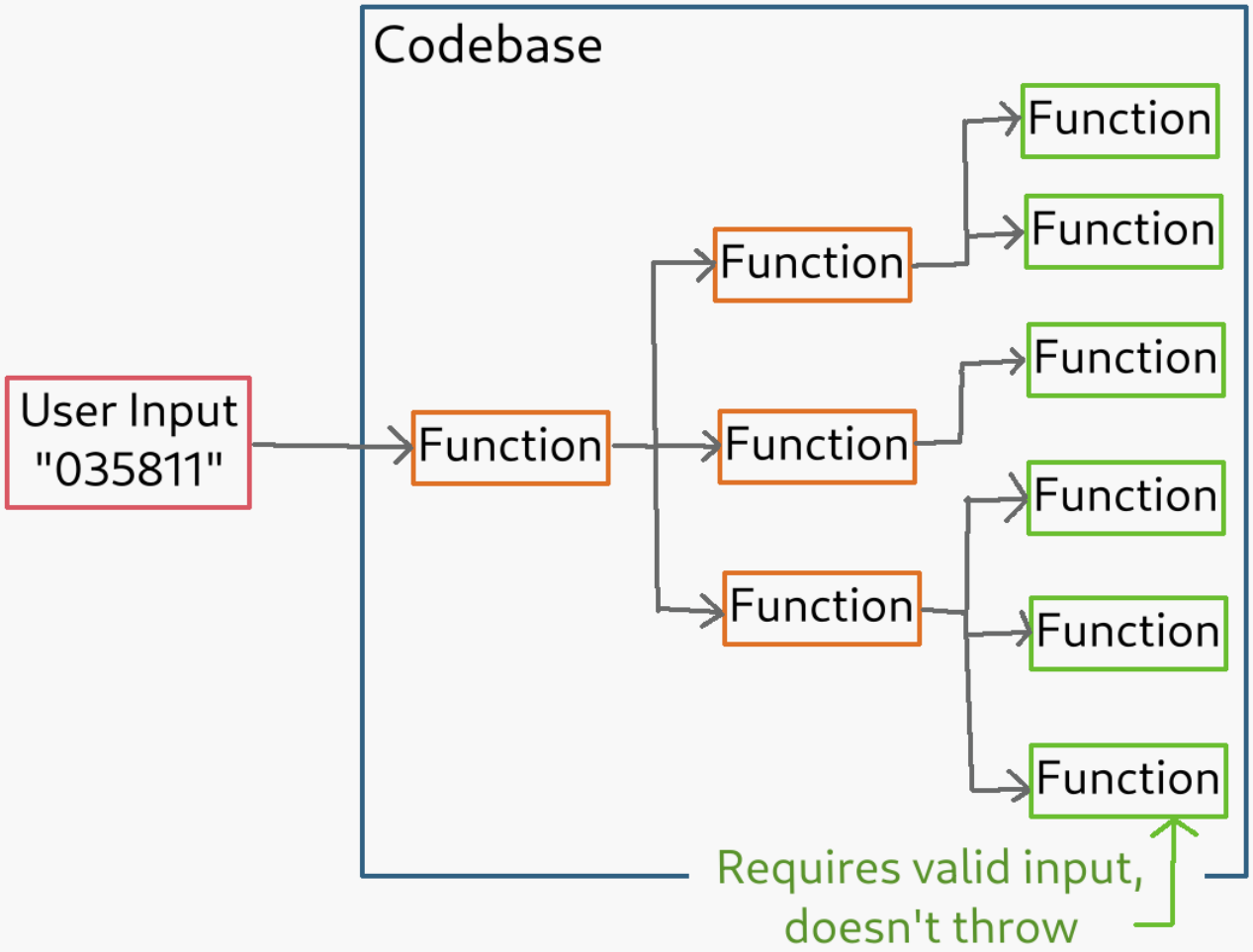
Problem: everything is a String

Example

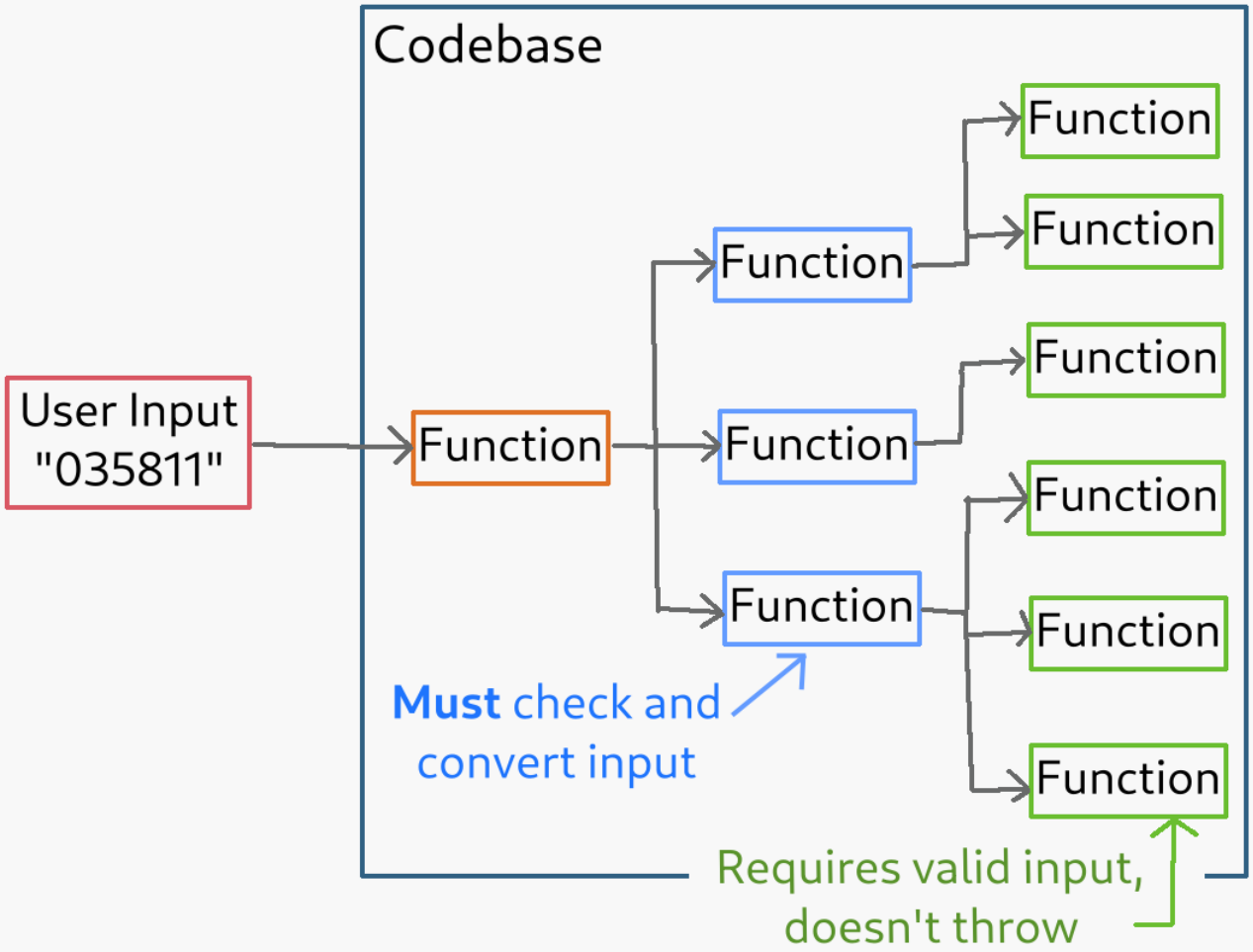


Solution: replace String with PostalCode incrementally

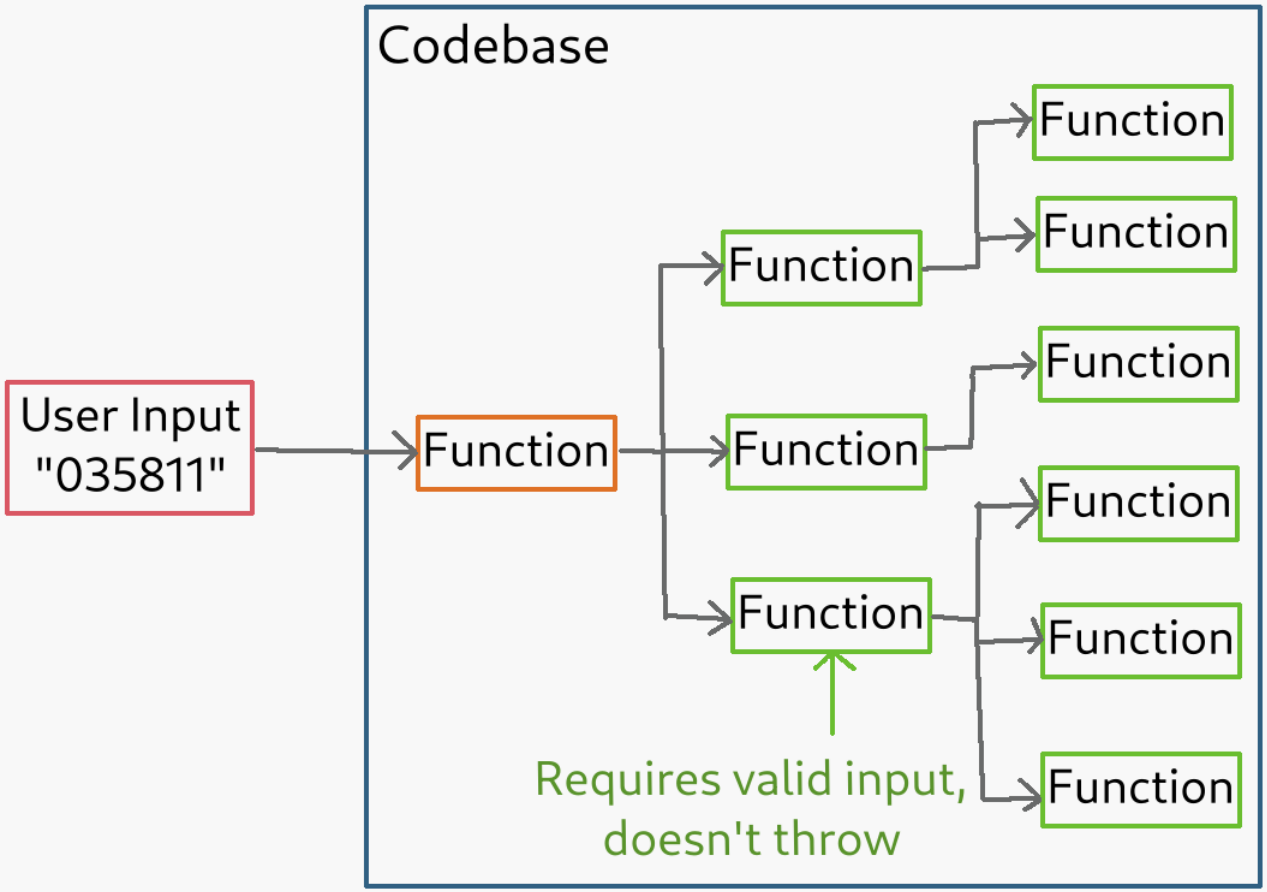
Example


Codebase
User Input
"035811"
Function
Function
Function
Function
Function
Function
Function
Function
Function
Function
Requires valid input,
doesn't throw

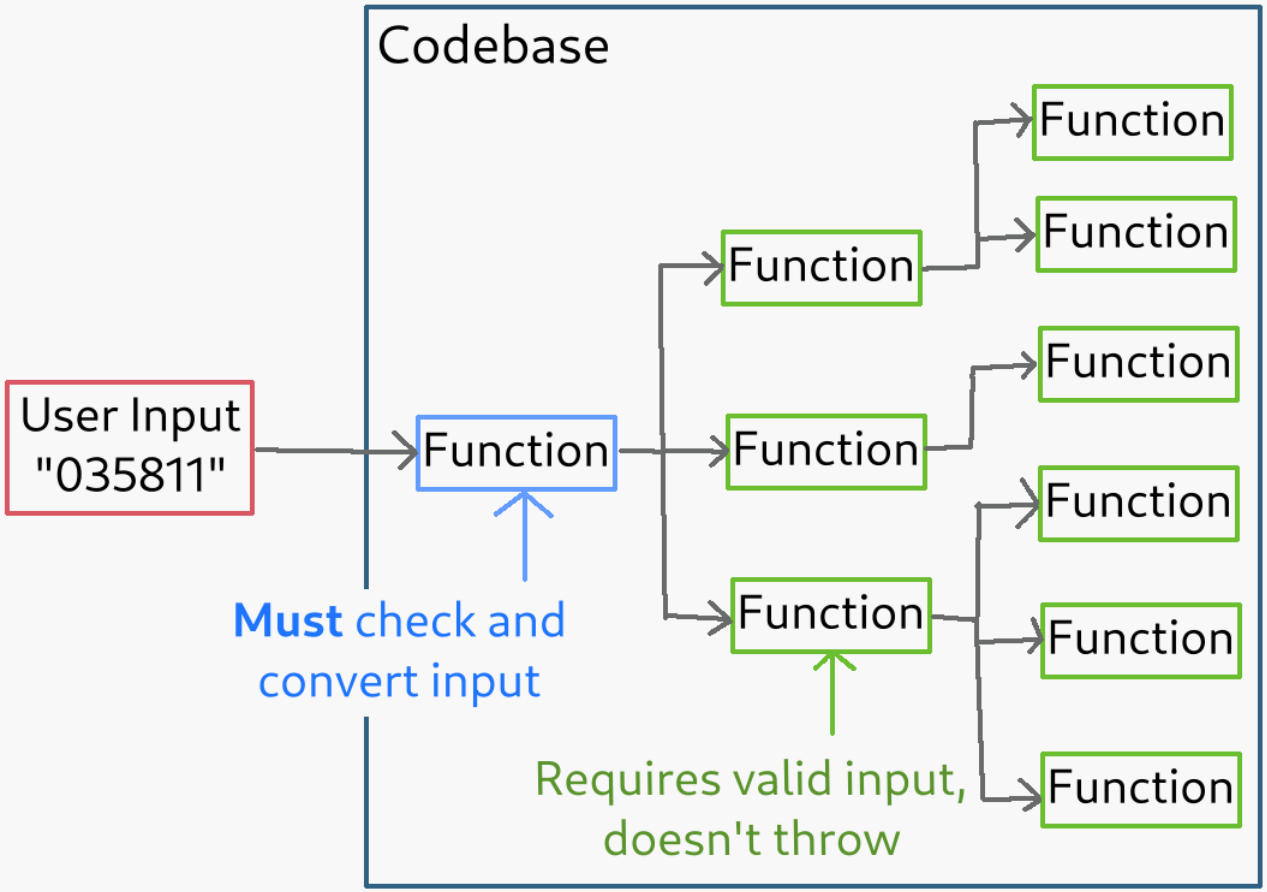
Example


Codebase
User Input
"035811"
Function
Function
Function
Function
Function
Function
Function
Function
Function
Function
Must check and
convert input
Requires valid input,
doesn't throw

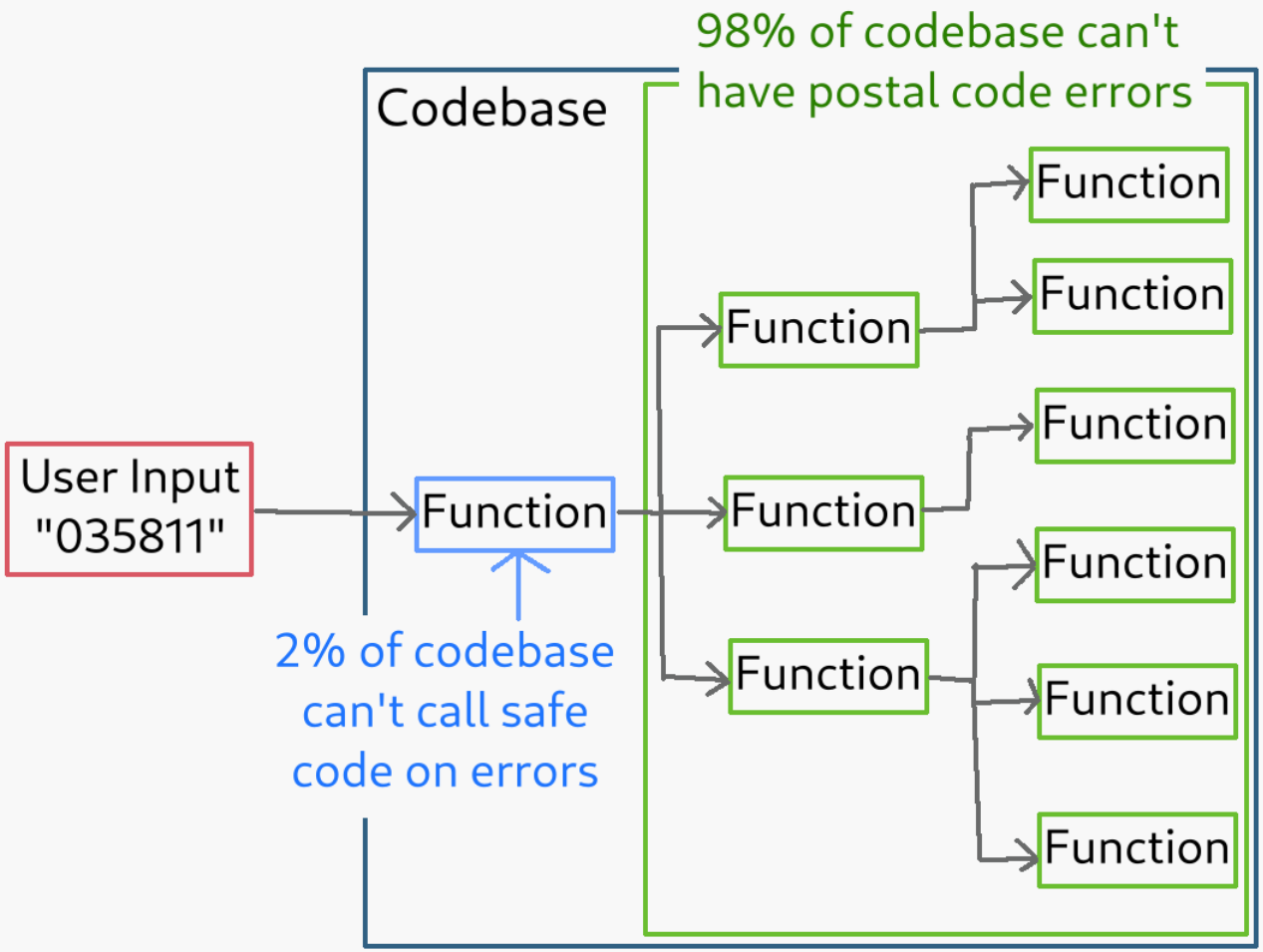
Example


Codebase
User Input
"035811"
Function
Function
Function
Function
Function
Function
Function
Function
Function
Function
Requires valid input,
doesn't throw

Example


Codebase
User Input
"035811"
Function
Function
Function
Function
Function
Function
Function
Function
Function
Function
Must check and
convert input
Requires valid input,
doesn't throw

Example


98% of codebase can't have postal code errors
Codebase
Function
Function
Function
Function
Function
Function
Function
Function
Function
User Input "035811"
2% of codebase can't call safe code on errors

Example

- Useful for runtime errors which pervade the entire codebase
- Most famous problem of this kind: null

Getting rid of null

- Problem: null objects, null pointers, null exceptions
- Solution: languages which support non-nullable types

Getting rid of null

- Example: C++ references
- Like pointers, but way more restrictive

Getting rid of null

- Example: C++ references
- Like pointers, but way more restrictive

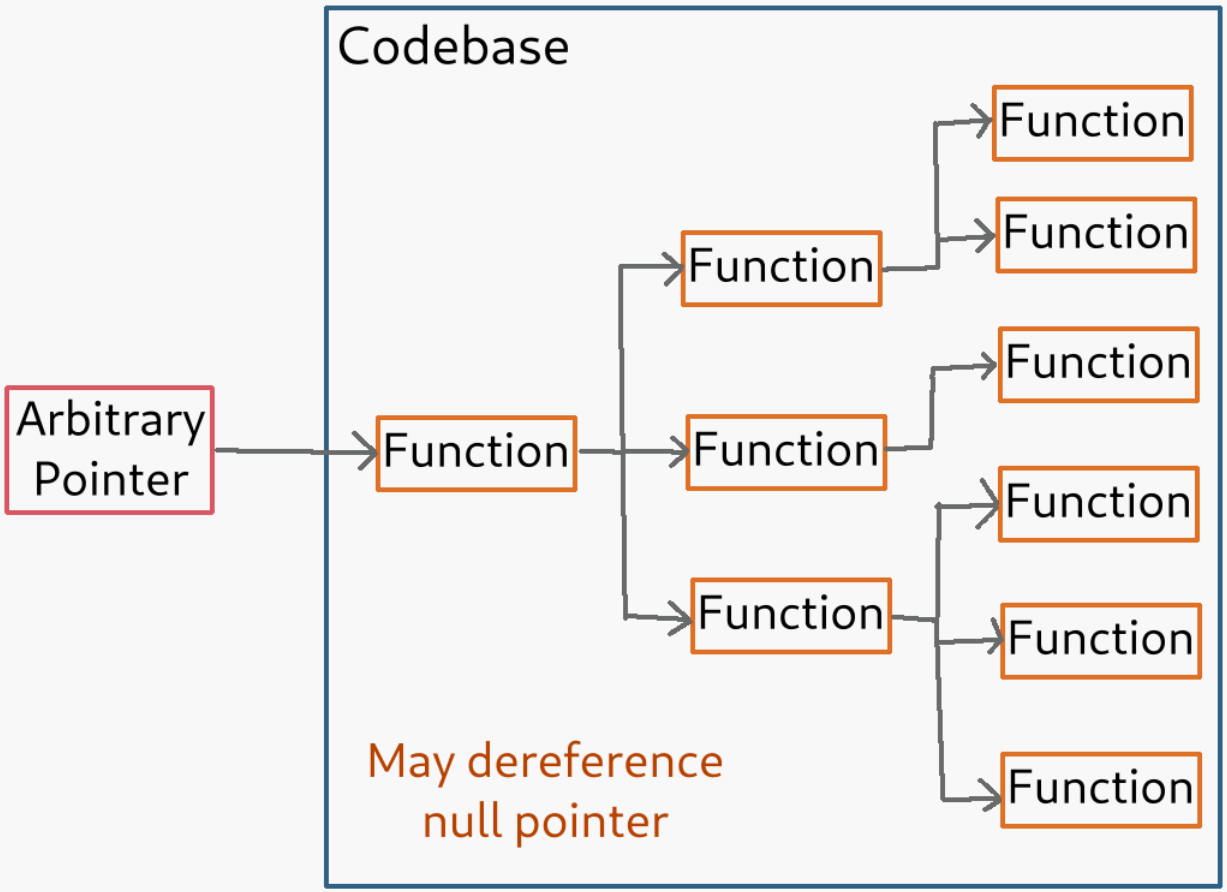
```
void setValue(string &value) { // reference
  value = "Test 123";
}
```

Getting rid of null

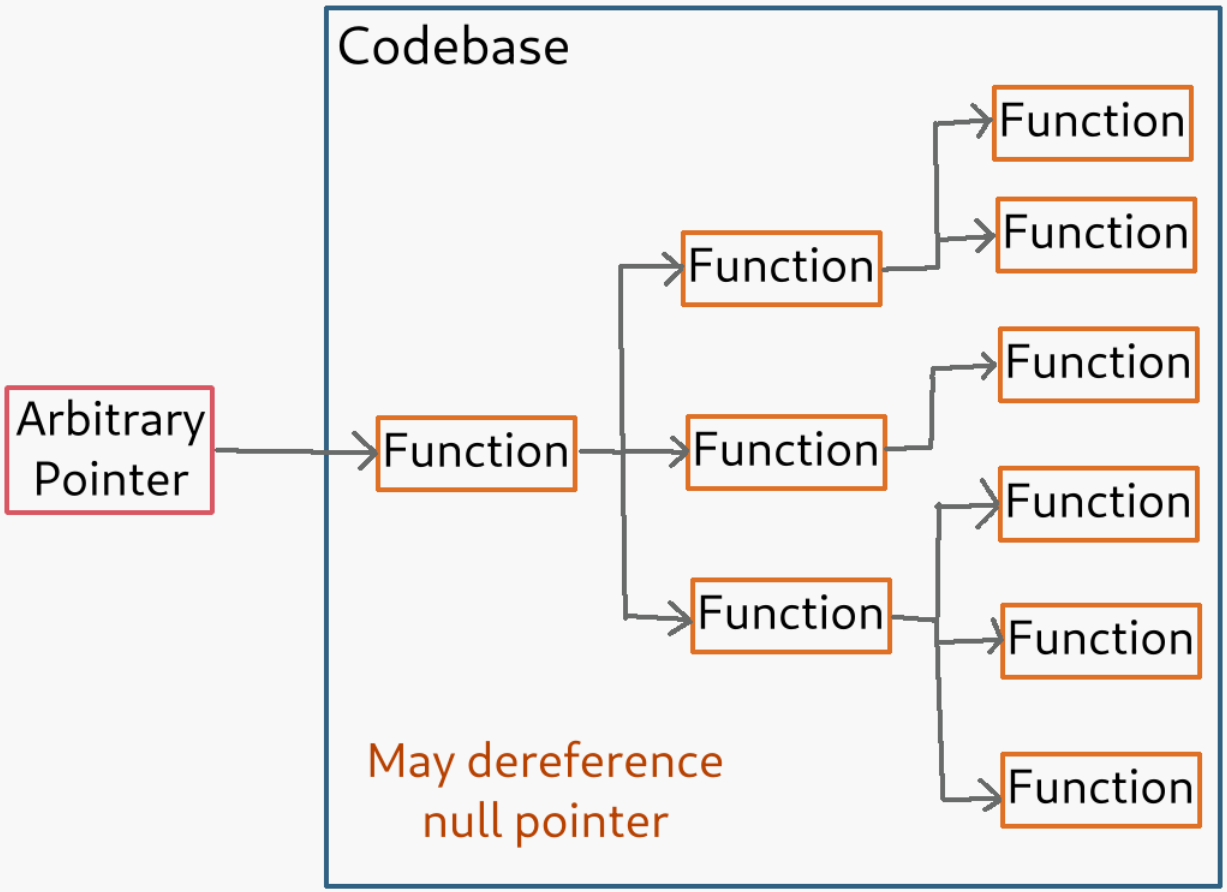
- Example: C++ references
- Like pointers, but way more restrictive

```
void setValue(string &value) { // reference
  value = "Test 123";
}

string *value = getName();

setValue(*value); // problem caused by caller,
                  // not by the function itself
```

Getting rid of null

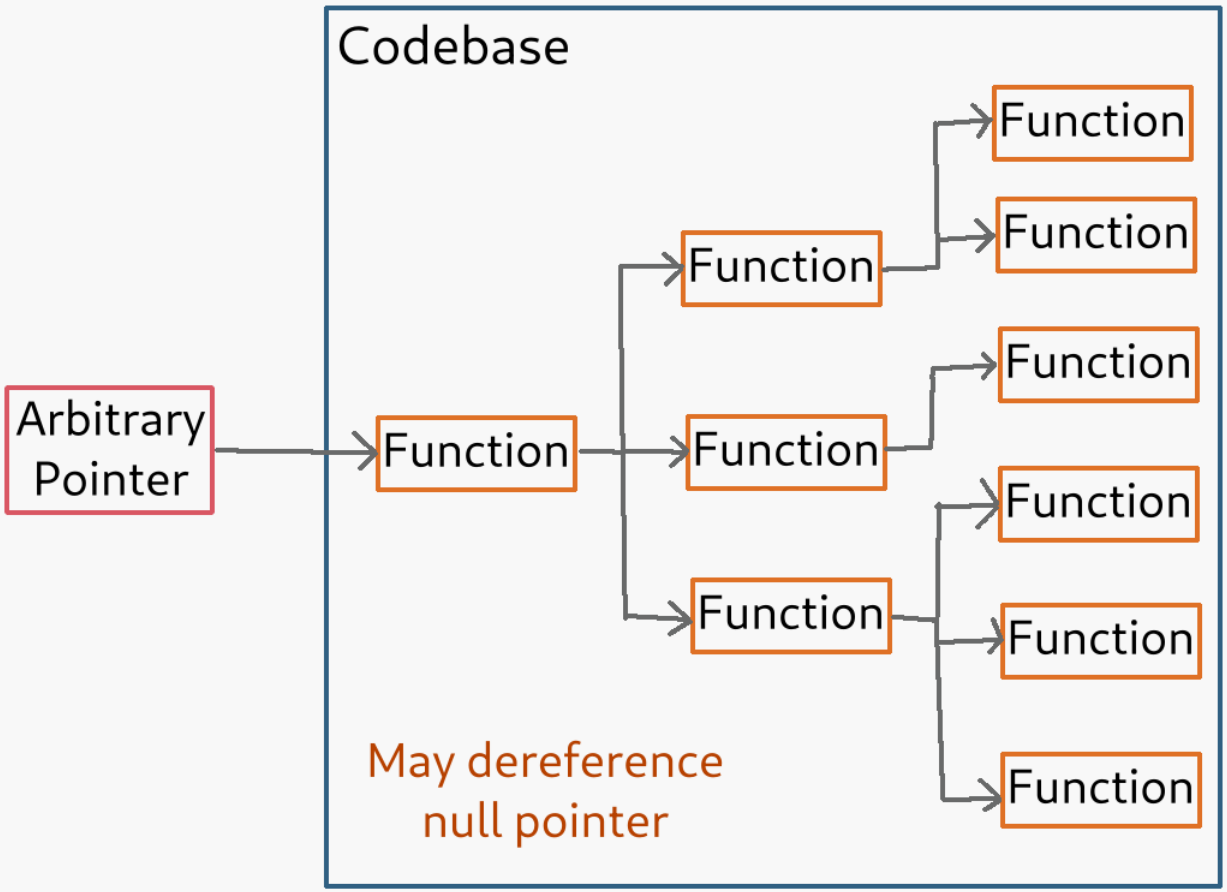


Getting rid of null



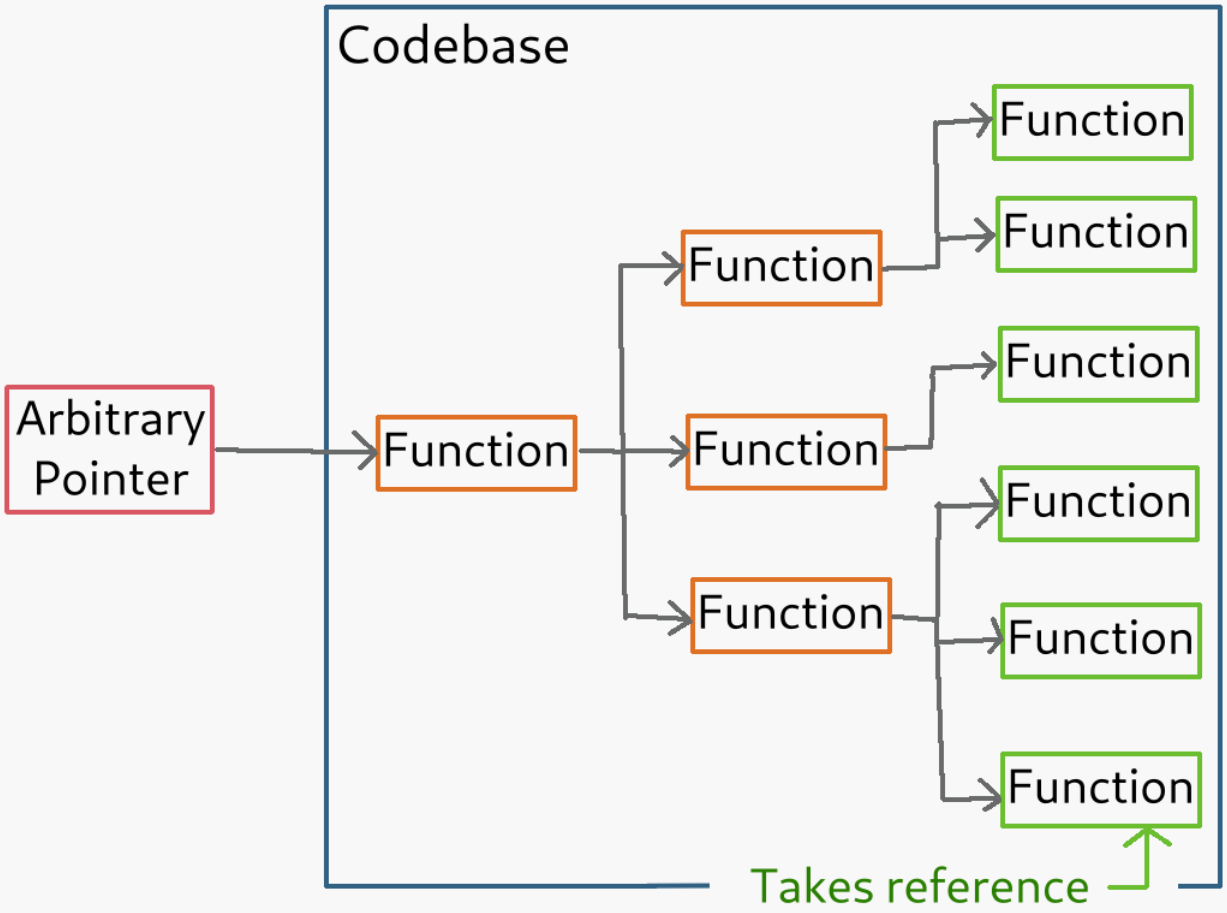
Problem: everything takes a pointer

Getting rid of null

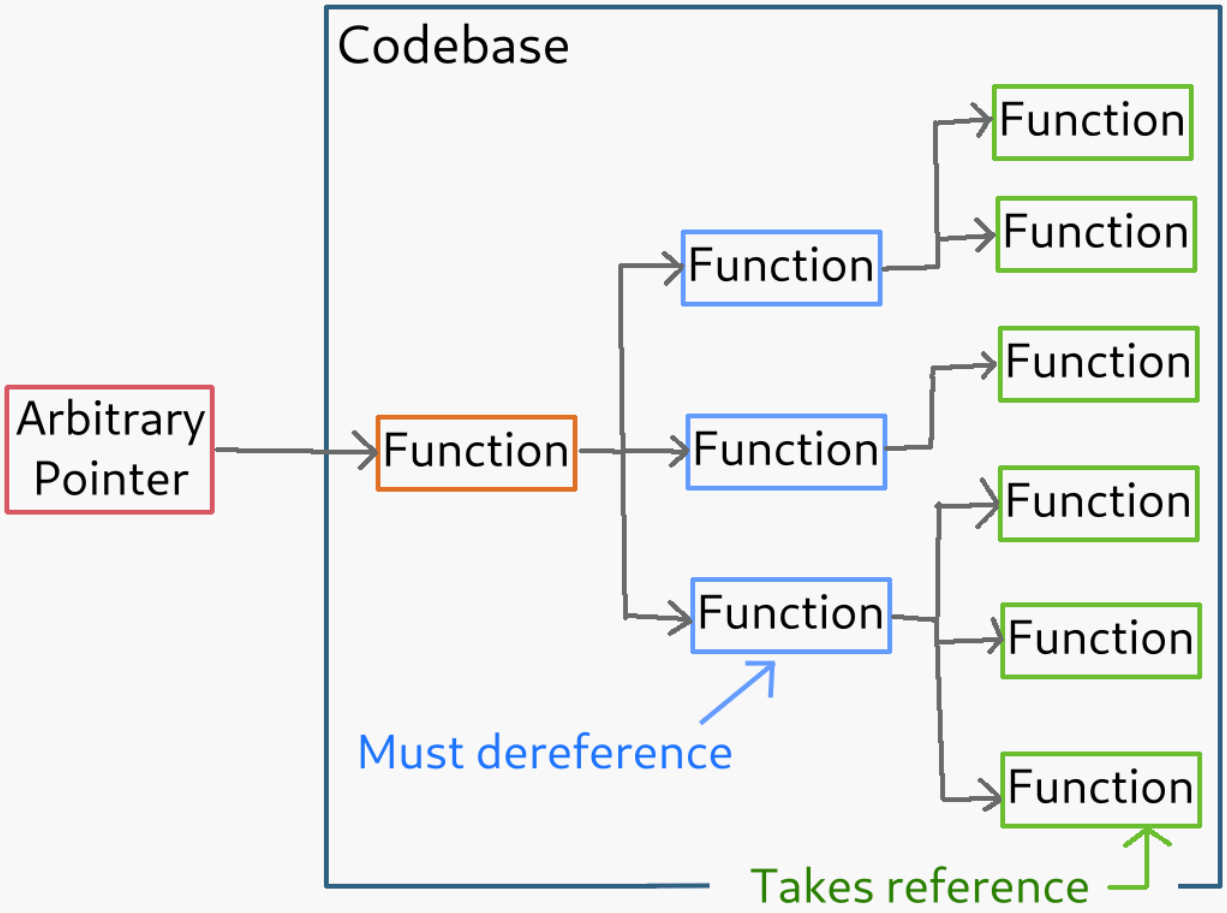


Solution: replace pointers with references

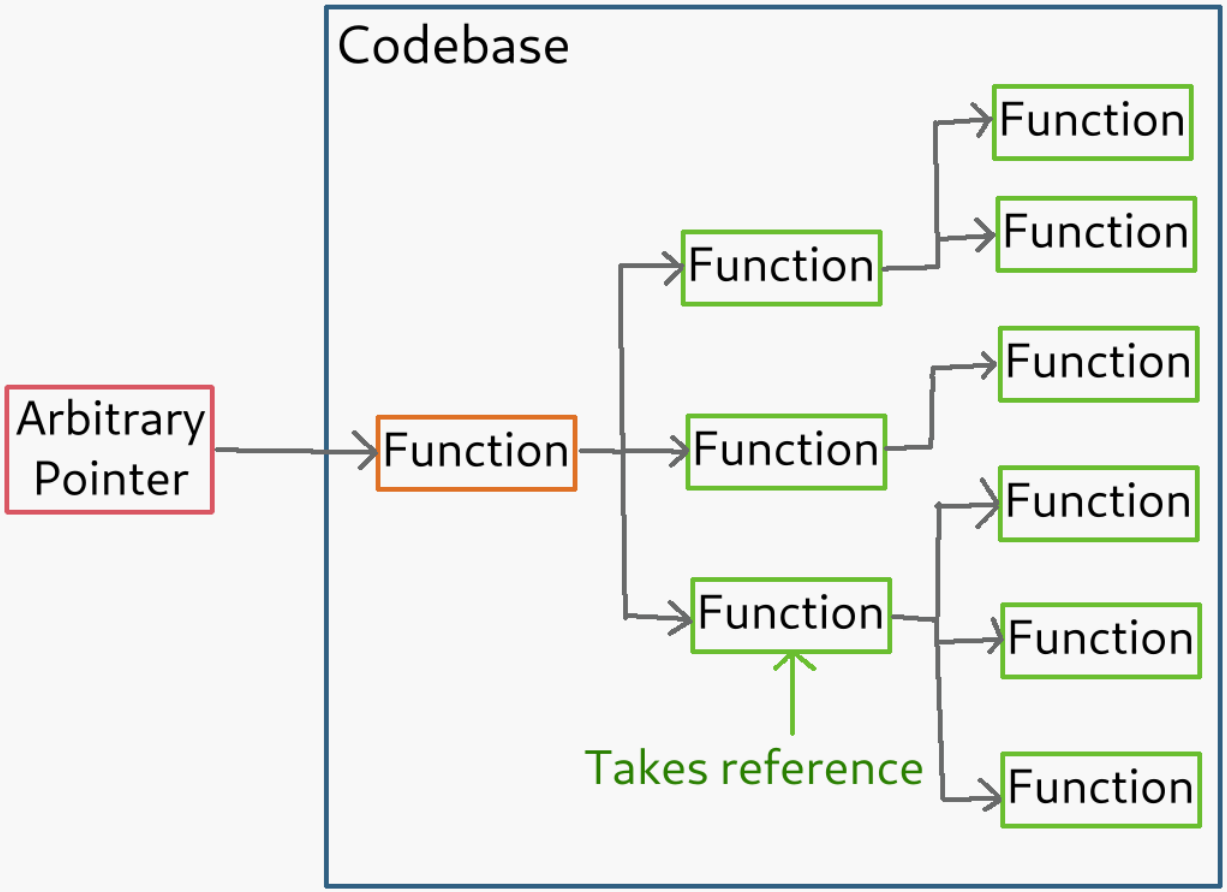
Getting rid of null



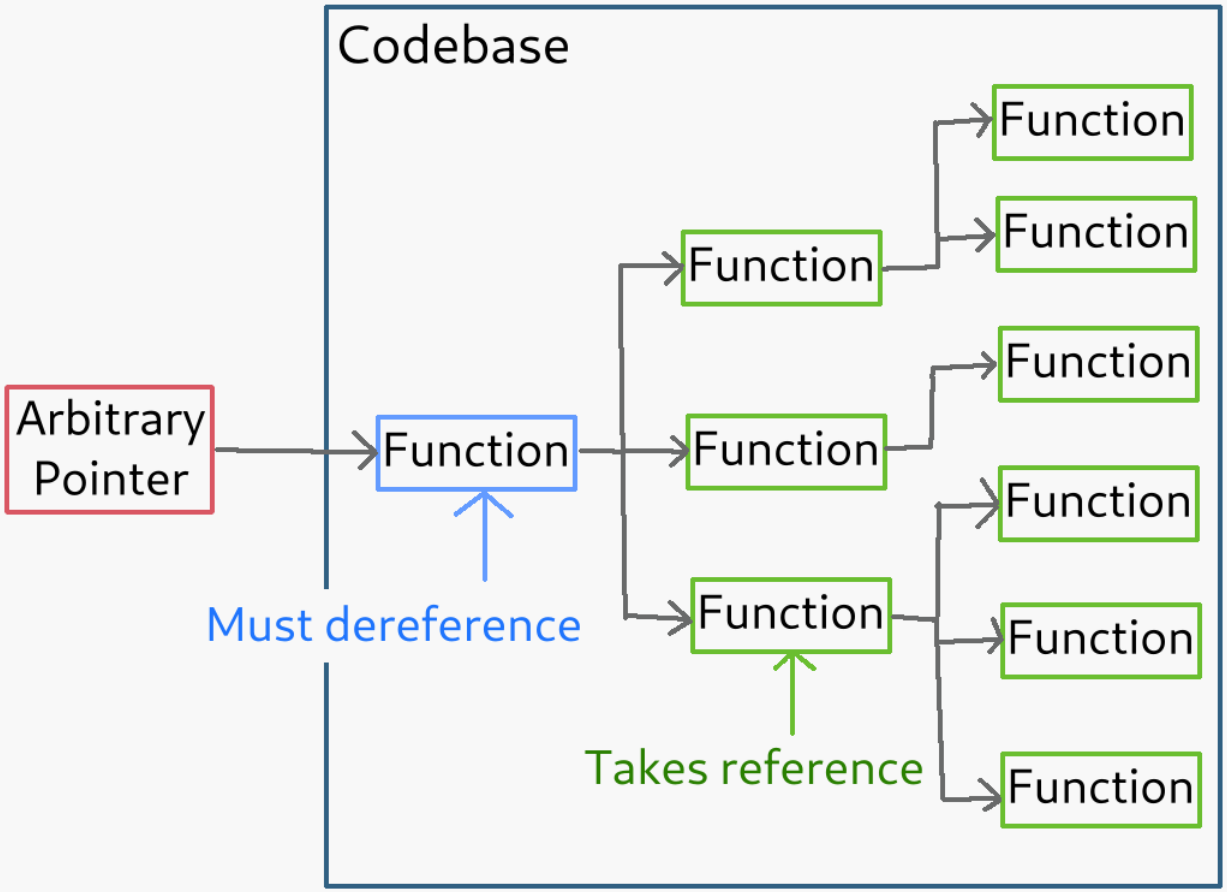
Getting rid of null



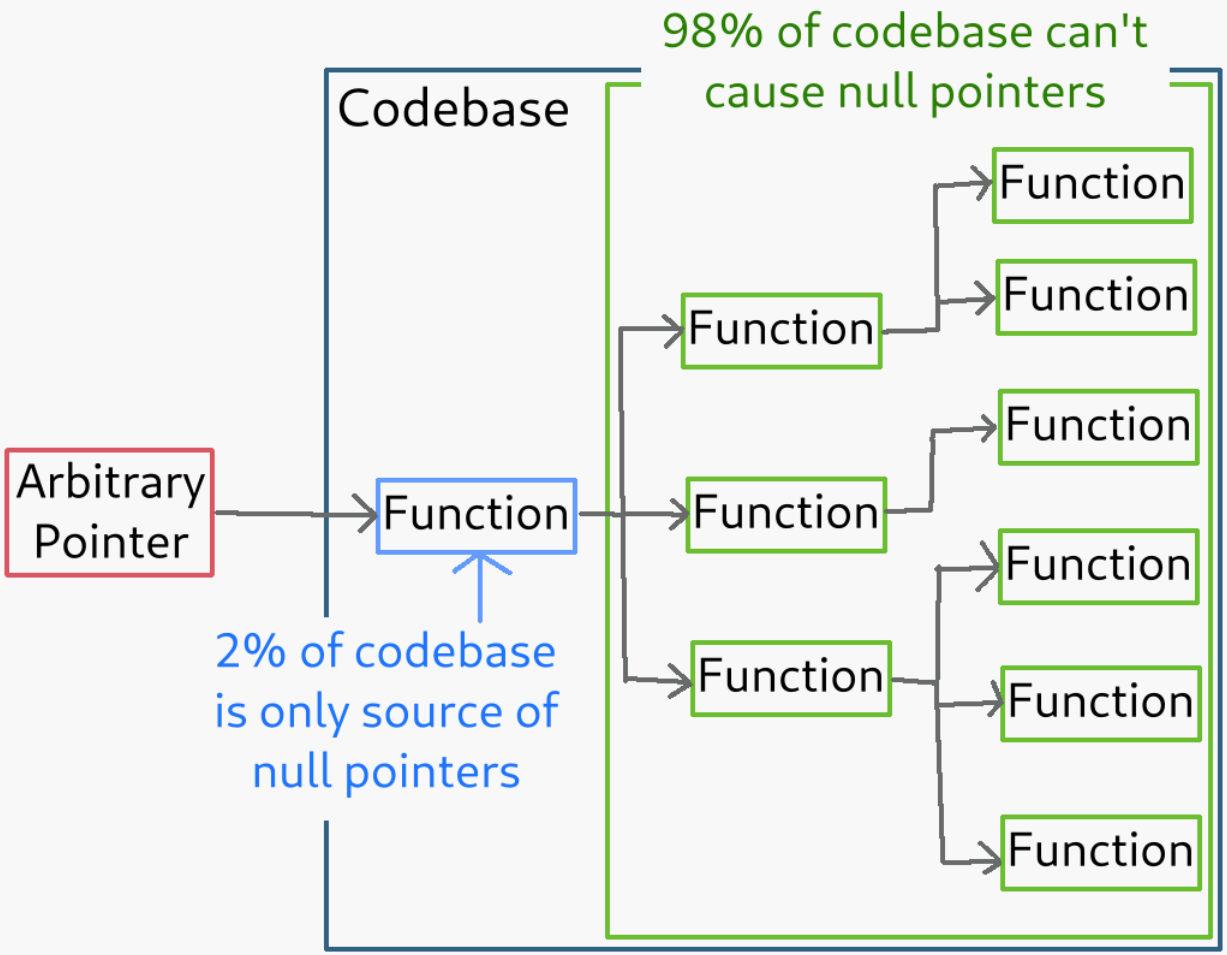
Getting rid of null



Getting rid of null



Getting rid of null



Getting rid of null

- How to fix the remaining 2%?

Getting rid of null

- How to fix the remaining 2%?
- Ban unsafe features by using tools, CI and code reviews
  - Risky casts
  - Raw pointer usage
  - Unchecked dereferencing of pointers and optionals

Getting rid of null

- How to fix the remaining 2%?
- Ban unsafe features by using tools, CI and code reviews
  - Risky casts
  - Raw pointer usage
  - Unchecked dereferencing of pointers and optionals
- Other languages make these errors completely impossible

Getting rid of null

- Pointers are not useless
- Its just that nullable objects are rarely wanted

Simplifying code

Simplifying code

```
/** Opens a new RepoWriter for safe writing into the specified repository.
  The returned RepoWriter will keep a reference to all the arguments passed
  to this function, so make sure not to free or modify them as long as the
  writer is in use. The caller of this function must ensure, that only one
  writer exists for the given repository. Otherwise it will lead to
  corrupted data.

  @param repo_path The path to the repository.
  @param repo_tmp_file_path The path to the dummy file inside the
  repository. This is the file to which all the data will be written. Once
  the writer gets closed, the data will be synced to disk and the dummy
  file gets renamed to the final file. If the dummy file already exists, it
  will be overwritten. The dummy file must be either inside the repository
  or on the same device as the repository in order to be effective.
  @param source_file_path The path to the original file, that gets written
  to the repository trough this writer. This is only needed in case of an
  error, to display which file failed to be written to the repository.
  @param info Informations describing the file, which gets written to the
  repository. This is needed for generating the filename inside the
  repository. All values inside this struct must be defined, otherwise it
  will lead to unexpected behaviour. So make sure, that the files size is
  larger than FILE_HASH_SIZE.

  @return A new RepoWriter, which must be closed by the caller using
  repoWriterClose().
*/
RepoWriter *repoWriterOpenFile(String repo_path,
                               String repo_tmp_file_path,
                               String source_file_path,
                               const RegularFileInfo *info)
{
  RepoWriter *writer = createRepoWriter(repo_path, repo_tmp_file_path,
                                        source_file_path, false);
  writer->rename_to.info = info;

  return writer;
}
```

← Lots of stuff to keep in mind when calling this function

Simplifying code

1. Having to keep many assumptions in the back of your head increases mental load

Simplifying code

1. Having to keep many assumptions in the back of your head increases mental load
2. Each function/class adds more assumptions

Simplifying code

1. Having to keep many assumptions in the back of your head increases mental load
2. Each function/class adds more assumptions
3. Exploding complexity makes these assumptions unmanageable

Simplifying code

1. Having to keep many assumptions in the back of your head increases mental load
2. Each function/class adds more assumptions
3. Exploding complexity makes these assumptions unmanageable
4. This causes people to give up and start guessing

Simplifying code

1. Having to keep many assumptions in the back of your head increases mental load
2. Each function/class adds more assumptions
3. Exploding complexity makes these assumptions unmanageable
4. This causes people to give up and start guessing
5. Result: trial-and-error programming which creates even more hidden assumptions

Simplifying code

- Types can express assumptions
- Compilers and IDEs understand types
- Result: accidental mistakes will be highlighted by your IDE

Simplifying code

- Whether an assumption can be represented by a type or not depends *highly* on the programming language
- If the language supports it, it should be utilized

Examples

Problem: null is not wanted

```
// ... [function doc] ...
// Never pass null to this function! PLEASE!
function processValue(String *value) {
   ...
}
```

Examples

Solution: use non-nullable types, value types, references

```
function processValue(String value) {
   ...
}
```

Examples

Problem: invalid input is not wanted

```
// ... [function doc] ...
// Only pass valid X509 PEM strings! PLEASE!
function processCert(String pem_encoded_x509_cert) {
   ...
}
```

Examples

Solution: use more specific and restrictive types

```
function processCert(X509Cert my_cert) {
   ...
}
```

Examples

Problem: resource has to be cleaned up

```
// ... [function doc] ...
// Release the returned handle with destroy()! PLEASE!
SomeHandle makeNewHandle() {
   ...
}
```

Examples

Solution: use types representing ownership

```
SomeHandle makeNewHandle() {
   ...
}
```

Examples

Problem: resource gets consumed by function

```
// ... [function doc] ...
// Never pass the same handle twice to this function!
function finalizeFile(LockFileHandle *handle) {
  ...
}
```

Examples

Solution: use types which can have only one owner

```
function finalizeFile(UniqueLockFileHandle handle) {
  ...
}
```

Examples

Problem: function is expected to fail during normal usage

```
// ... [function doc] ...
// PLEASE catch these exceptions: <long exception list>
String readFile(String path) {
  ...
}
```

Examples

Solution: option types. Similar to null, but more restrictive. IDE hints and autocomplete make it obvious that this function may not return a string

```
Optional<String> readFile(String path) {
   ...
}
```

Examples

Problem: function is expected to fail/succeed in too many ways

```
// ... [function doc] ...
// Check the return code, out parameters, exceptions...
int receiveData(String *name_out, PostalCode *code_out) {
  ...
}
```

Examples

Solution: use variants, sum types, union types

```
type Result = ErrorCode | String | PostalCode;

Result receiveData() {
   ...
}
```

Examples

Solution: use variants, sum types, union types

```
type Result = ErrorCode | String | PostalCode;

Result receiveData() {
  ...
}
match(receiveData()) {
  case ErrorCode  as e:    ...
  case String     as name: ...
  case PostalCode as code  ...
}
```

Examples

Solution: use variants, sum types, union types

```
type Result = ErrorCode | String | PostalCode;

Result receiveData() {
  ...
}
match(receiveData()) {
  case ErrorCode  as e:    ...
  case String     as name: ...
  case PostalCode as code  ...
}
```

Mandatory matching makes possible results obvious

Examples

Problem: unit mismatch

```
int time        = 1500;
int weight      = 80;
int temperature = 93;

int result = time * weight + temperature;
```

Examples

Solution: use stronger types

```
Seconds time        = 1500;
Kilo    weight      = 80;
Celsius temperature = 93;

int result = time * weight + temperature;
```

Examples

Solution: use stronger types

```
Seconds time        = 1500;
Kilo    weight      = 80;
Celsius temperature = 93;

int result = time * weight + temperature;
```

↑ compiler: invalid unit conversion

Examples

Solution: use stronger types

```
Seconds time        = 1500;
Kilo    weight      = 80;
Celsius temperature = 93;

int result = time * weight + temperature;
```

↑ compiler: invalid unit conversion

```
Meter distance = 20;
KMh speed = distance / time;
```

↑ valid overload

Examples

- There is so much more which can be expressed with types
- Guarantees provided by types are enforced by the compiler
- Guarantees can be combined

Examples

```
class CipherContext {
  constructor();

  bool init(Byte[] entropy);
  void setId(String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
  constructor();                          To be called first,
                                        only once, and fails
  bool init(Byte[] entropy);                    on bad entropy
  void setId(String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
  constructor();

  bool init(Byte[] entropy);
  void setId(String id);
                                        To be called second,
  PublicKey getPubkey();                   may fail. What if
  bool completeHandshake(PublicKey peer); done twice?

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
  constructor();

  bool init(Byte[] entropy);
  void setId(String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);
                          Needs success of previous calls
  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
  constructor();                    Problem: How to simplify?

  bool init(Byte[] entropy);
  void setId(String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
  constructor();
  bool init(Byte[] entropy);
  void setId(String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Solution: Merge into constructor

Examples

```
class CipherContext {

  constructor(Byte[] entropy, String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {

  // Throws on bad entropy
  constructor(Byte[] entropy, String id);


  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
                                                    How to fix?
  // Throws on bad entropy                          (if wanted)
  constructor(Byte[] entropy, String id);


  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {
 ...                          Solution: new type which can
                                 only be instantiated from
                                  reliable entropy sources
class HighQualityEntropy {
  ...
}
```

- Known at runtime: whether entropy source is available
- Known at compile time: instantiation fails if not available

Examples

```
class CipherContext {
                                                    Can't fail

  constructor(HighQualityEntropy seed, String id);


  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class CipherContext {

  constructor(HighQualityEntropy seed, String id);

  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

How to fix?

Examples

```
class CipherContext {

  constructor(HighQualityEntropy seed, String id);

                                Solution: split class in two
  PublicKey getPubkey();
  bool completeHandshake(PublicKey peer);

  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class Handshake {
  constructor(HighQualityEntropy seed, String id);

  PublicKey getPubkey();
  Optional<Context> completeHandshake(PublicKey peer);
}

class Context {
  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
class Handshake {
  constructor(HighQualityEntropy seed, String id);

  PublicKey getPubkey();
  Optional<Context> completeHandshake(PublicKey peer);
}                    ↖ This is the entire trick

class Context {
  Byte[] encrypt(Byte[] plaintext);
  Byte[] decrypt(Byte[] ciphertext);
}
```

Examples

```
function doEncryption(Context context) {
   ...
}
```

Examples

```
function doEncryption(Context context) {
   ...
}
```

If this function ever gets called we will know that

- the handshake was successful
- seeded with high entropy
- done in the right order
- done only once

Refactoring

What if assumptions change?

Refactoring

```
/** Opens a new RepoWriter for safe writing into the specified repository.
  The returned RepoWriter will keep a reference to all the arguments passed
  to this function, so make sure not to free or modify them as long as the
  writer is in use. The caller of this function must ensure, that only one
  writer exists for the given repository. Otherwise it will lead to
  corrupted data.

  @param repo_path The path to the repository.
  @param repo_tmp_file_path The path to the dummy file inside the
  repository. This is the file to which all the data will be written. Once
  the writer gets closed, the data will be synced to disk and the dummy
  file gets renamed to the final file. If the dummy file already exists, it
  will be overwritten. The dummy file must be either inside the repository
  or on the same device as the repository in order to be effective.
  @param source_file_path The path to the original file, that gets written
  to the repository trough this writer. This is only needed in case of an
  error, to display which file failed to be written to the repository.
  @param info Informations describing the file, which gets written to the
  repository. This is needed for generating the filename inside the
  repository. All values inside this struct must be defined, otherwise it
  will lead to unexpected behaviour. So make sure, that the files size is
  larger than FILE_HASH_SIZE.

  @return A new RepoWriter, which must be closed by the caller using
  repoWriterClose().
*/
RepoWriter *repoWriterOpenFile(String repo_path,
                               String repo_tmp_file_path,
                               String source_file_path,
                               const RegularFileInfo *info)
{
  RepoWriter *writer = createRepoWriter(repo_path, repo_tmp_file_path,
                                        source_file_path, false);
  writer->rename_to.info = info;

  return writer;
}
```

Add more comments and hope somebody reads them?

Refactoring

Solution: change the types to represent new assumptions

Refactoring

Solution: change the types to represent new assumptions

1. Code breaks in the right places
2. Compiler error list becomes a todo list
3. Compiler points out what to fix
4. Modified old code shows up in the merge request to be reviewed

Runtime errors

- Not every error can be caught statically
- Some assumptions must be enforced at runtime
- Violated runtime assumptions must fail fast and loud
- Unit tests can catch breakage of runtime assumptions
- Deeply nested code makes testing much harder