Lifetime analysis in Clang

RFC

Utkarsh Saxena, Google

usx@google.com

Current State

Statement Local analysis

```cpp
std::string_view bound(std::string_view s [[clang::lifetimebound]]) {
    return s;
}

void stmt_local() {
    std::string_view result = bound(std::string());
                                //  ^^^^^^^^^^^^^
                                // error: pointer to temporary string object.
}
```

Current State

No Full Function analysis

```
void function_local() {
    std::string_view result;
    {
        std::string small = "small";
        result = bound(small);  // No error here.
    }
    std::cout << result;        // use-after-scope!
}
```

C++ Lifetime Model: An alias-based approach

Loans and OriginSets

```
void simple() {
    std::string_view ptr; // ptr's origin set 'O is {} (empty)
    {
        std::string small = "short lived";
        ptr = small; // Taking a reference of 'small '=> Loan 'L'
                     // 'O = {L}.
    }
    // Loan L expires.
    // 'O is {<expired L>}
    std::cout << ptr; // use-after-scope
}
```

C++ Lifetime Model: An alias-based approach

Flow sensitive

```cpp
void branch(bool condition) {
    std::string large = "long lived";
    std::string_view ptr = large; // 'O = {L_large}

    if (condition) {
        std::string small = "short lived";
        ptr = small; // 'O = {L_small}
    }
    // Origin sets merge: 'O = {L_large, L_small}
    std::cout << ptr; // potential use-after-scope
}
```

Permissive and Strict modes

```
std::string global_str = "STATIC";

std::string_view permissive() {
  std::string local = "local";
  view = local;  // 'local' doesn't live long enough [-Wdangling-safety-permissive]
  return view;
}

std::string_view strict(bool condition) {
  std::string_view view = global_str;
  std::string local = "local";
  if (condition) {
    view = local;  // error: 'local' doesn't live long enough [-Wdangling-safety]
  }
  return view;
}
```

Future enhancements

1. Annotation Suggestions

```cpp
std::string_view Identity(std::string_view in) {
  return in;  // warning: Add [[lifetimebound]] on 'in'
}
```

Future enhancements

2. Annotation Verifications

```
std::string Copy(std::string_view in [[clang::lifetimebound]]) {
             // ^ warning: lifetimebound param 'in' is not returned.
    std::string out = std::string(in);
    return out;
}
```

Future enhancements

3. Iterator invalidation

Exclusivity for some opt-in types.

```
void foo() {
    std::vector<int> v = {1, 2, 3};
    auto it = v.begin();
    v.push_back(4); // error: modifying 'v' invalidated 'it'.
    std::cout << (*it);
}
```

Thank you

Reachout on RFC/discord.