# How To Argue(ment)

Richard Powell

v1.4

# The function signature

```
void f(???);
```

- The function signature is your “thesis statement".
- Different argument types mean different things.
- Think about it as the caller and the callee.

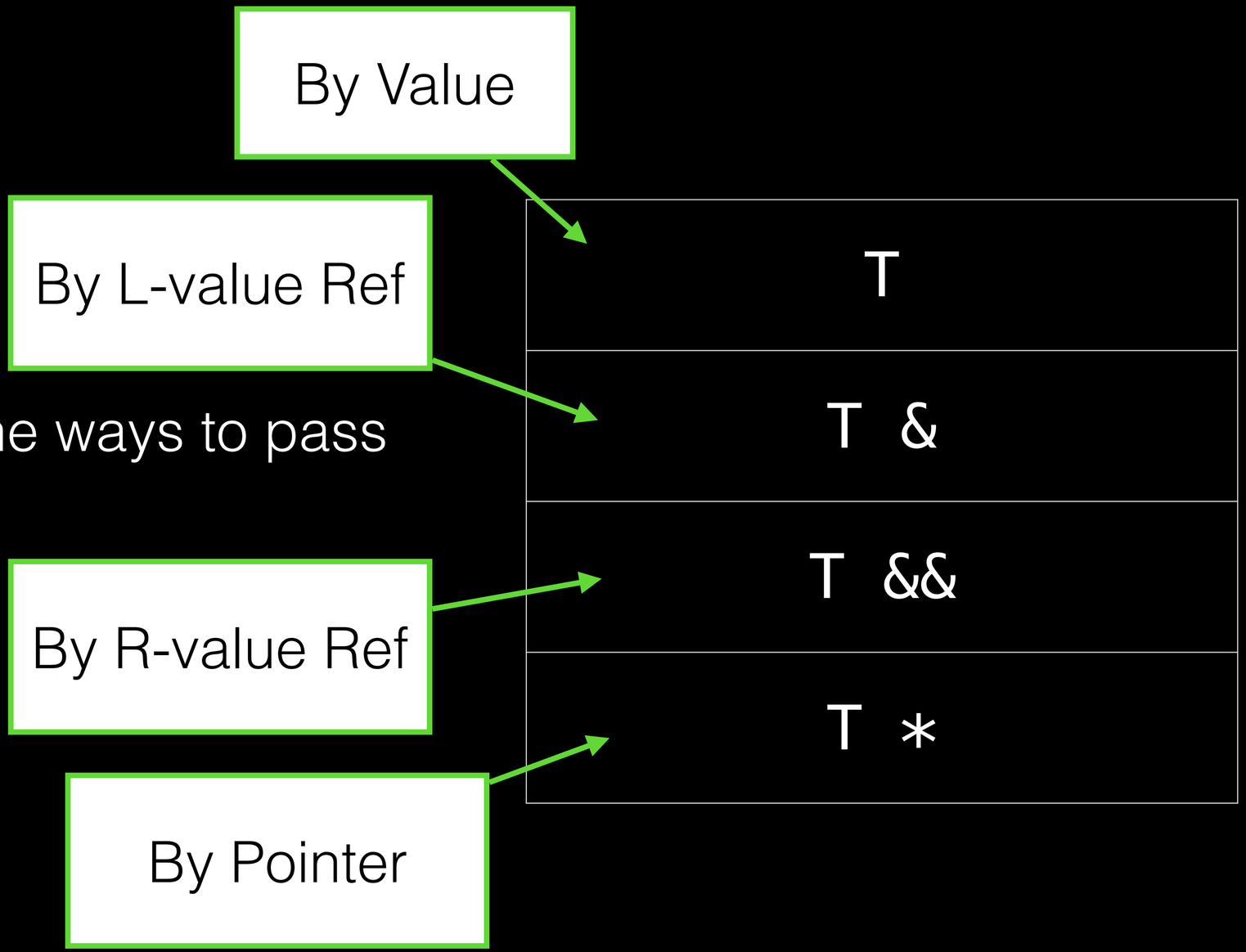
# Best Practices

```
void f(???);
```

- Express argument’s purpose via type.
- Use the most “generic” type possible.
- Don’t surprise your callers.

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?



```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | `T const` |
| `T &` | `T const &` |
| `T &&` | `T const &&` |
| `T *` | `T * const` |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | `T const` |
| `T &` | `T const &` |
| `T &&` | `T const &&` |
| `T *` | `T * const` |
| `T * &` | `T * const &` |
| `T * &&` | `T * const &&` |
| `T const *` | `T const * const` |
| `T const * &` | `T const * const &` |
| `T const * &&` | `T const * const &&` |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| T | T const |
|---|---|
| T & | T const & |
| T && | T const && |
| T * | T * const |
| T * & | T * const & |
| T * && | T * const && |
| T const * | T const * const |
| T const * & | T const * const & |
| T const * && | T const * const && |

| unique_ptr<T> | unique_ptr<T> const |
|---|---|
| unique_ptr<T> & | unique_ptr<T> const & |
| unique_ptr<T> && | unique_ptr<T> const && |
| unique_ptr<T const> | unique_ptr<T const> const |
| unique_ptr<T const> & | unique_ptr<T const> const & |
| unique_ptr<T const> && | unique_ptr<T const> const && |

| shared_ptr<T> | shared_ptr<T> const |
|---|---|
| shared_ptr<T> & | shared_ptr<T> const & |
| shared_ptr<T> && | shared_ptr<T> const && |
| shared_ptr<T const> | shared_ptr<T const> const |
| shared_ptr<T const> & | shared_ptr<T const> const & |
| shared_ptr<T const> && | shared_ptr<T const> const && |

# Pass by value

```
void foo(T);
```

- Snapshot

```
void foo(T);
```

vs

```
void foo(T const);
```

- `const` here is for the callee, not the caller.
- Compiler views them as the same

# Pass by `const` reference

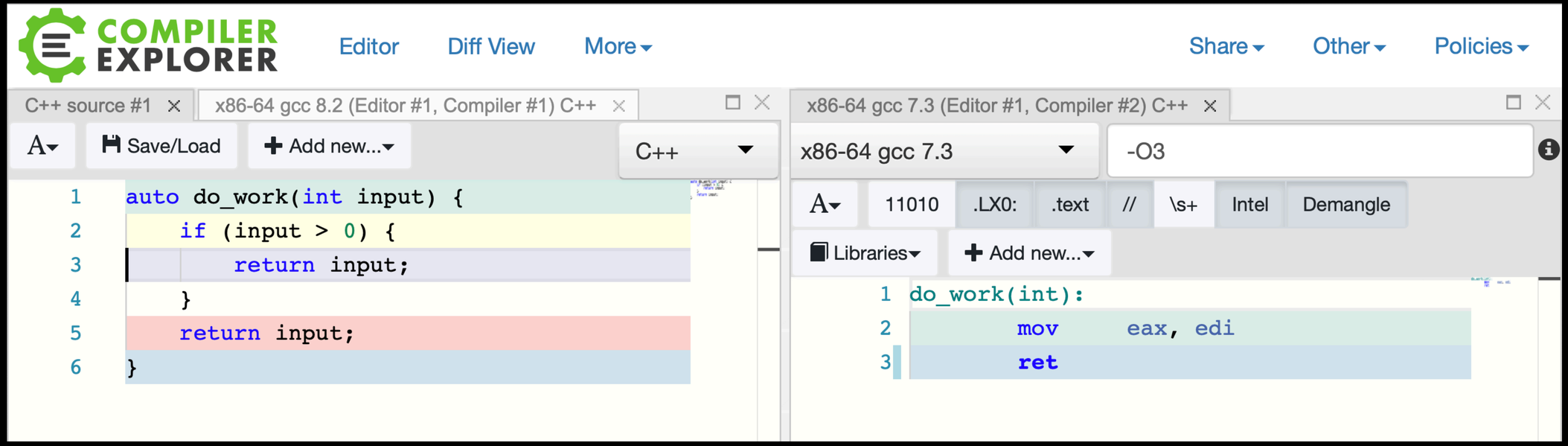
```
void foo(T const &);
```

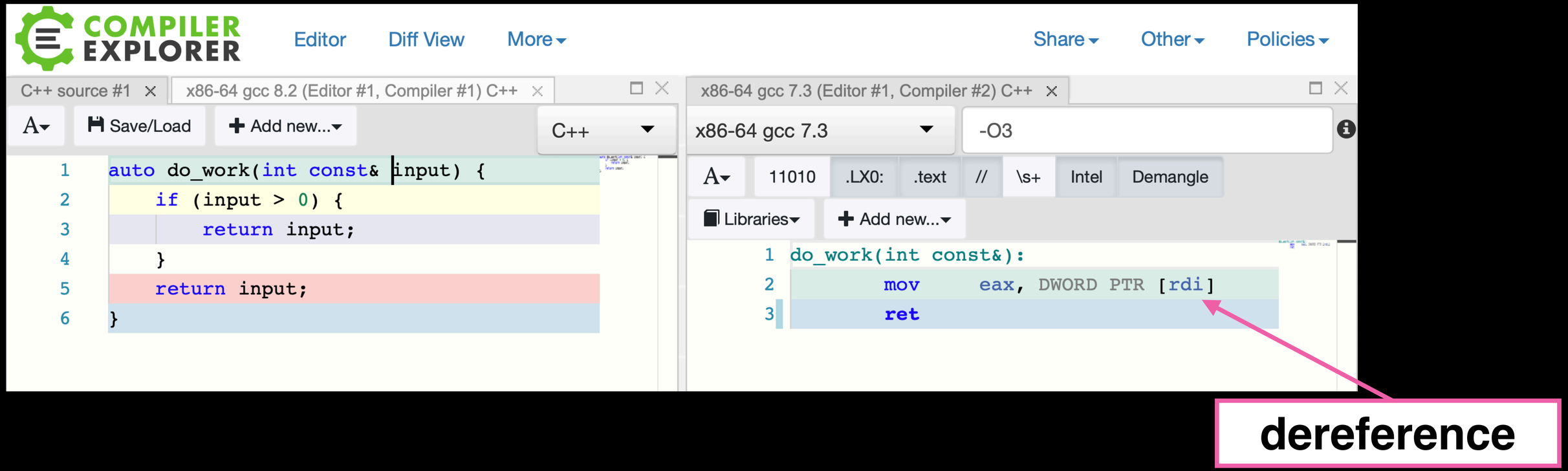
- Observe (during call)

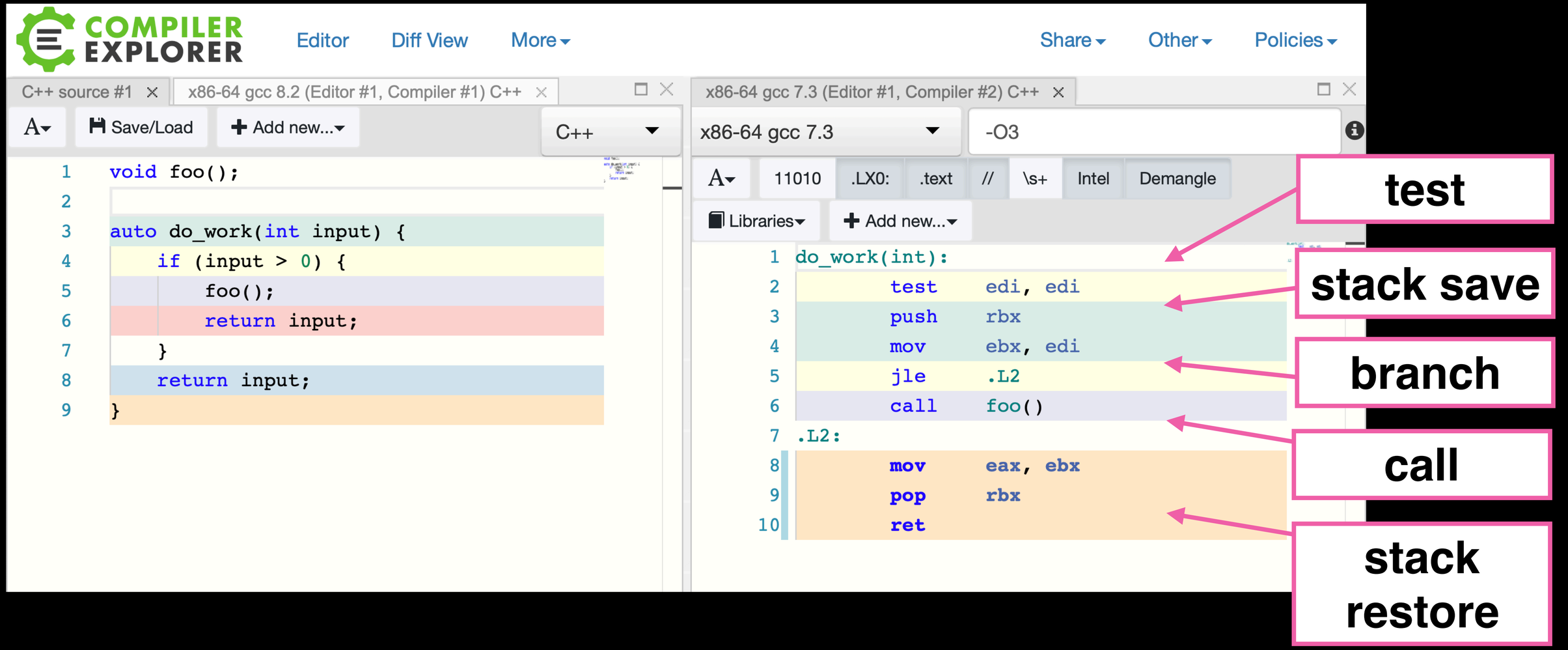
```
void foo(T);
```

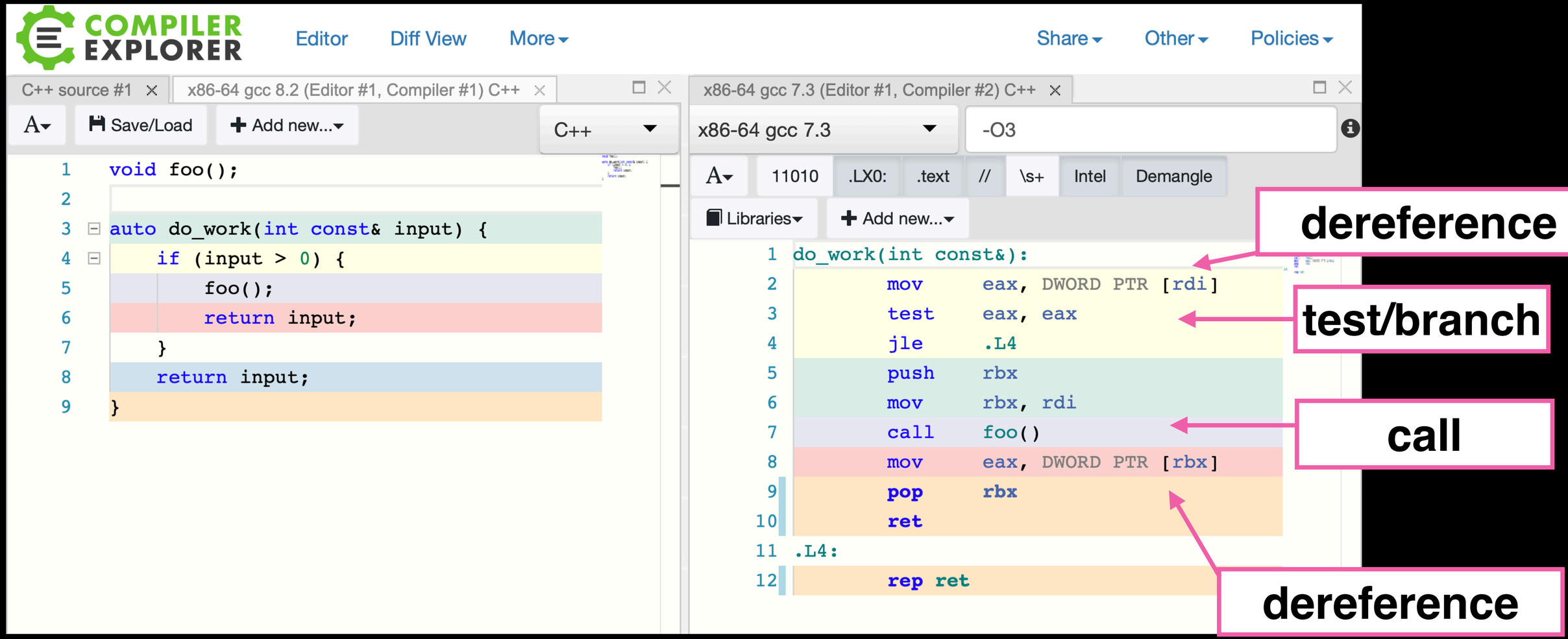
vs

```
void foo(T const &);
```


COMPILER EXPLORER
Editor
Diff View
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 gcc 7.3 (Editor #1, Compiler #2) C++
Save/Load
Add new...
C++
x86-64 gcc 7.3
-O3
11010
.LX0:
.text
//
\s+
Intel
Demangle
Libraries
Add new...
1 auto do_work(int input) {
2 if (input > 0) {
3 return input;
4 }
5 return input;
6 }
1 do_work(int):
2 mov eax, edi
3 ret


COMPILER EXPLORER
Editor
Diff View
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 gcc 7.3 (Editor #1, Compiler #2) C++
Save/Load
Add new...
C++
x86-64 gcc 7.3
-O3
11010
.LX0:
.text
//
\s+
Intel
Demangle
Libraries
Add new...
1 auto do_work(int const& input) {
2 if (input > 0) {
3 return input;
4 }
5 return input;
6 }
1 do_work(int const&):
2 mov eax, DWORD PTR [rdi]
3 ret
dereference


COMPILER EXPLORER
Editor
Diff View
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 gcc 7.3 (Editor #1, Compiler #2) C++
Save/Load
Add new...
C++
x86-64 gcc 7.3
-O3
11010
.LX0:
.text
//
\s+
Intel
Demangle
Libraries
Add new...
void foo();
auto do_work(int input) {
if (input > 0) {
foo();
return input;
}
return input;
}
do_work(int):
test edi, edi
push rbx
mov ebx, edi
jle .L2
call foo()
.L2:
mov eax, ebx
pop rbx
ret
test
stack save
branch
call
stack restore


COMPILER EXPLORER
Editor
Diff View
More
Share
Other
Policies
C++ source #1
x86-64 gcc 8.2 (Editor #1, Compiler #1) C++
x86-64 gcc 7.3 (Editor #1, Compiler #2) C++
Save/Load
Add new...
C++
x86-64 gcc 7.3
-O3
11010
.LX0:
.text
//
\s+
Intel
Demangle
Libraries
Add new...
1 void foo();
2
3 auto do_work(int const& input) {
4 if (input > 0) {
5 foo();
6 return input;
7 }
8 return input;
9 }
1 do_work(int const&):
2 mov eax, DWORD PTR [rdi]
3 test eax, eax
4 jle .L4
5 push rbx
6 mov rbx, rdi
7 call foo()
8 mov eax, DWORD PTR [rbx]
9 pop rbx
10 ret
11 .L4:
12 rep ret
dereference
test/branch
call
dereference

```
int a;
void foo() {
  ++a;
}

int do_work(int const& input) {
  if (input > 0) {
    foo();
    return input;
  }
  return input;
}

int bar() {
  return do_work(a);
}
```

```
void foo(T);
```

vs

```
void foo(T const &);
```

- Pass by value when you can, pass by const reference when you must.

# Pass by reference

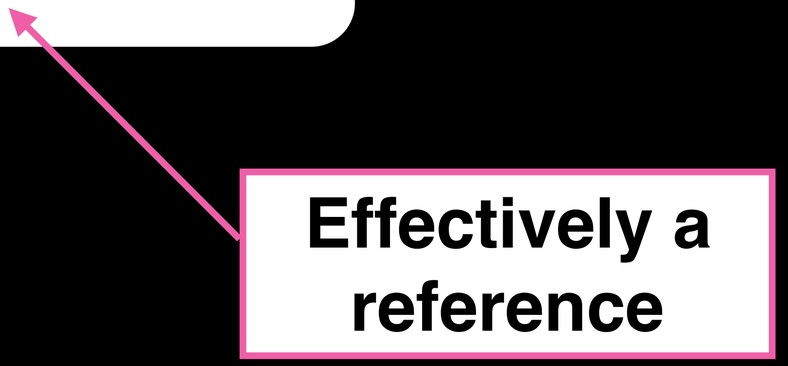
```
void foo(T &);
```

- Modify (during call)

- “I would never do that!”

```
void Cat::declaw();
```

```
void _Cat_declaw(Cat* this);
```



# Pass by R-Value reference

```
void foo(T &&);
```

- Sink

# Pass by `const` R-Value reference

```
void foo(T const &&);
```

- R-values make sense for objects that can be moved from.
- Moved means “steal” or “disembowel”.
- `const&&` doesn’t make sense.

# Pass by (`const`) pointer

```
void foo(T *);
```

```
void foo(T const *);
```

- Optional (during call)
  - **NEVER** as an output parameter value
  - **Never for lifetime management!**

```
void foo(T *);
```

vs

```
void foo(std::optional<T>);
```

- Which is more general?

```
void foo(int *);

int a = ans();
foo(&a);
```

vs

```
void foo(std::optional<int>);

int a = ans();
foo(a);
```

Passing a thing

```
void foo(int *);

foo(nullptr);
```

vs

```
void foo(std::optional<int>);

foo(nullptr); // ERROR
```

Passing not a thing

```
void foo(int *);

foo(nullptr);
```

vs

```
void foo(std::optional<int>);

foo({});
```

Passing not a thing

```
void foo(int * t)
{
  *t = 3;
}

int i = 1;
foo(i);
assert(i == 3);
```

vs

```
void foo(std::optional<int> t)
{
  *t = 3;
}

int i = 1;
foo(i);
assert(i == 3); // ERROR
```

```
void foo(std::mutex const*);

std::mutex m;
foo(&m);
```

```
void foo(std::optional<std::mutex const>);

std::mutex m;
foo(m); // ERROR
```

```
void foo(T *);
```

vs

```
void foo(std::optional<T>);
```

- Prefer by Pointer to express Optional

# Pass by `const` L-value Ref to (`const`) pointer

```
void foo(T * const &);
```

```
void foo(T const * const &);
```

- Taking a `const&` to a pointer doesn't make sense.

# Pass by R-value Ref to (`const`) pointer

```
void foo(T * &&);
```

```
void foo(T const * &&);
```

- T* is just a pointer, a primitive value.
- An R-Value of a primitive is just a copy.  Doesn't make sense.

# Pass by L-value Ref to (`const`) pointer

```
void foo(T * &);
```

```
void foo(T const * &);
```

- NEVER do this!
- Effectively “reseating” a pointer.
- Interferes with Lifetime management

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | `T const &&` |
| `T *` | |
| `T * &` | `T * const &` |
| `T * &&` | `T * const &&` |
| `T const *` | |
| `T const * &` | `T const * const &` |
| `T const * &&` | `T const * const &&` |

| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | `unique_ptr<T> const &` |
| `unique_ptr<T> &&` | `unique_ptr<T> const &&` |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | `unique_ptr<T const> const &` |
| `unique_ptr<T const> &&` | `unique_ptr<T const> const &&` |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| `shared_ptr<T> &&` | `shared_ptr<T> const &&` |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| `shared_ptr<T const> &&` | `shared_ptr<T const> const &&` |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | |
| `T *` | |
| `T * &` | `T * const &` |
| `T * &&` | |
| `T const *` | |
| `T const * &` | `T const * const &` |
| `T const * &&` | |

| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | `unique_ptr<T> const &` |
| `unique_ptr<T> &&` | |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | `unique_ptr<T const> const &` |
| `unique_ptr<T const> &&` | |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| `shared_ptr<T> &&` | |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| `shared_ptr<T const> &&` | |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | |
| `T *` | |
| `T * &` | |
| `T * &&` | |
| `T const *` | |
| `T const * &` | |
| `T const * &&` | |

| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | `unique_ptr<T> const &` |
| `unique_ptr<T> &&` | |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | `unique_ptr<T const> const &` |
| `unique_ptr<T const> &&` | |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| `shared_ptr<T> &&` | |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| `shared_ptr<T const> &&` | |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | |
| `T *` | |
| `T * &` | |
| → | |
| `T const *` | |
| `T const * &` | |
| → | |

| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | `unique_ptr<T> const &` |
| `unique_ptr<T> &&` | |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | `unique_ptr<T const> const &` |
| `unique_ptr<T const> &&` | |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| `shared_ptr<T> &&` | |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| `shared_ptr<T const> &&` | |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| T | |
| T & | T const & |
| T && | |
| T * | |
| → | |
| | |
| T const * | |
| → | |
| | |

| | |
|---|---|
| unique_ptr<T> | |
| unique_ptr<T> & | unique_ptr<T> const & |
| unique_ptr<T> && | |
| unique_ptr<T const> | |
| unique_ptr<T const> & | unique_ptr<T const> const & |
| unique_ptr<T const> && | |

| | |
|---|---|
| shared_ptr<T> | |
| shared_ptr<T> & | shared_ptr<T> const & |
| shared_ptr<T> && | |
| shared_ptr<T const> | |
| shared_ptr<T const> & | shared_ptr<T const> const & |
| shared_ptr<T const> && | |

# Lifetime Pointers

`unique_ptr<T>` `shared_ptr<T>`

- Transfer Ownership
- Share Ownership
- Optionally Share Ownership

# Pass by `unique_ptr<T>`

```
void foo(unique_ptr<T>);
```

```
void foo(unique_ptr<T const>);
```

- Transfer Ownership

# Pass by `const` L-value Ref to `unique_ptr<T>`

```
void foo(unique_ptr<T> const &);
```

```
void foo(unique_ptr<T const> const &);
```

```
// old C++98 code
struct Bar {
  static Bar* make_Bar(std::string);
};

void log_stats(Bar const*);



Bar* my_b = Bar::make_Bar("cat");
log_stats(my_b);
```

```
// new C++11 code
struct Bar {
  static std::unique_ptr<Bar> make_Bar(std::string);
};

void log_stats(Bar const*);



Bar* my_b = Bar::make_Bar("cat");
log_stats(my_b);
```

```
// new C++11 code
struct Bar {
  static std::unique_ptr<Bar> make_Bar(std::string);
};

void log_stats(Bar const*);


auto my_b = Bar::make_Bar("cat");
log_stats(my_b);
```

```
// new C++11 code
struct Bar {
  static std::unique_ptr<Bar> make_Bar(std::string);
};

void log_stats(std::unique_ptr<Bar> const&);



auto my_b = Bar::make_Bar("cat");
log_stats(my_b);
```

```
// new C++11 code
struct Bar {
  static std::unique_ptr<Bar> make_Bar(std::string);
};

void log_stats(Bar const*);


auto my_b = Bar::make_Bar("cat");
log_stats(my_b.get());
```

But raw pointers are bad, right?

*Fine to use for observing*

# Pass by `const` L-value Ref to `unique_ptr<T>`

```
void foo(unique_ptr<T> const &);
```

```
void foo(unique_ptr<T const> const &);
```

Use pointer instead:

```
void foo(T *);
```

```
void foo(T const *);
```

# Pass by R-value Ref `unique_ptr<T>`

```
void foo(unique_ptr<T> &&);
```

```
void foo(unique_ptr<T const> &&);
```

- `unique_ptr<T>` has to be passed by R-Value.
- `unique_ptr<T>&&` doesn't make sense.

# Pass by `shared_ptr<T>`

```
void foo(shared_ptr<T>);
```

```
void foo(shared_ptr<T const>);
```

- Share Ownership

```
void do_work(shared_ptr<Foo> input)
{
}
```

- Only use `shared_ptr<T>` if you are sharing ownership.

# Pass by `const` L-Value Ref `shared_ptr<T>`

```
void foo(shared_ptr<T> const &);
```

```
void foo(shared_ptr<T const> const &);
```

- Optionally Share Ownership

```
void do_work(shared_ptr<Foo> const & input)
{
  if (run_async) {
    start_backround_calc(input);
  }
  else {
    …
  }
}
```

# Pass by R-Value Ref `shared_ptr<T>`

```
void foo(shared_ptr<T> &&);
```

```
void foo(shared_ptr<T const> &&);
```

- Sink?
- use `shared_ptr<T>` instead.

# Pass by L-Value Ref smart pointer

```
void foo(unique_ptr<T> &);
```

```
void foo(unique_ptr<T const> &);
```

```
void foo(shared_ptr<T> &);
```

```
void foo(shared_ptr<T const> &);
```

- Reseat
- Rare

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| T | |
|---|---|
| T & | T const & |
| T && | |
| T * | |
| | |
| | |
| T const * | |
| | |
| | |

| unique_ptr<T> | |
|---|---|
| unique_ptr<T> & | |
| unique_ptr<T> && | |
| unique_ptr<T const> | |
| unique_ptr<T const> & | |
| unique_ptr<T const> && | |

| shared_ptr<T> | |
|---|---|
| shared_ptr<T> & | shared_ptr<T> const & |
| shared_ptr<T> && | |
| shared_ptr<T const> | |
| shared_ptr<T const> & | shared_ptr<T const> const & |
| shared_ptr<T const> && | |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | |
| `T *` | |
| | |
| | |
| `T const *` | |
| | |
| | |

| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | |
| | |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | |
| | |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| `shared_ptr<T> &&` | |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| `shared_ptr<T const> &&` | |

```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | |
| `T *` | |
| | |
| | |
| `T const *` | |
| | |
| | |

| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | |
| | |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | |
| | |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| | |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| | |

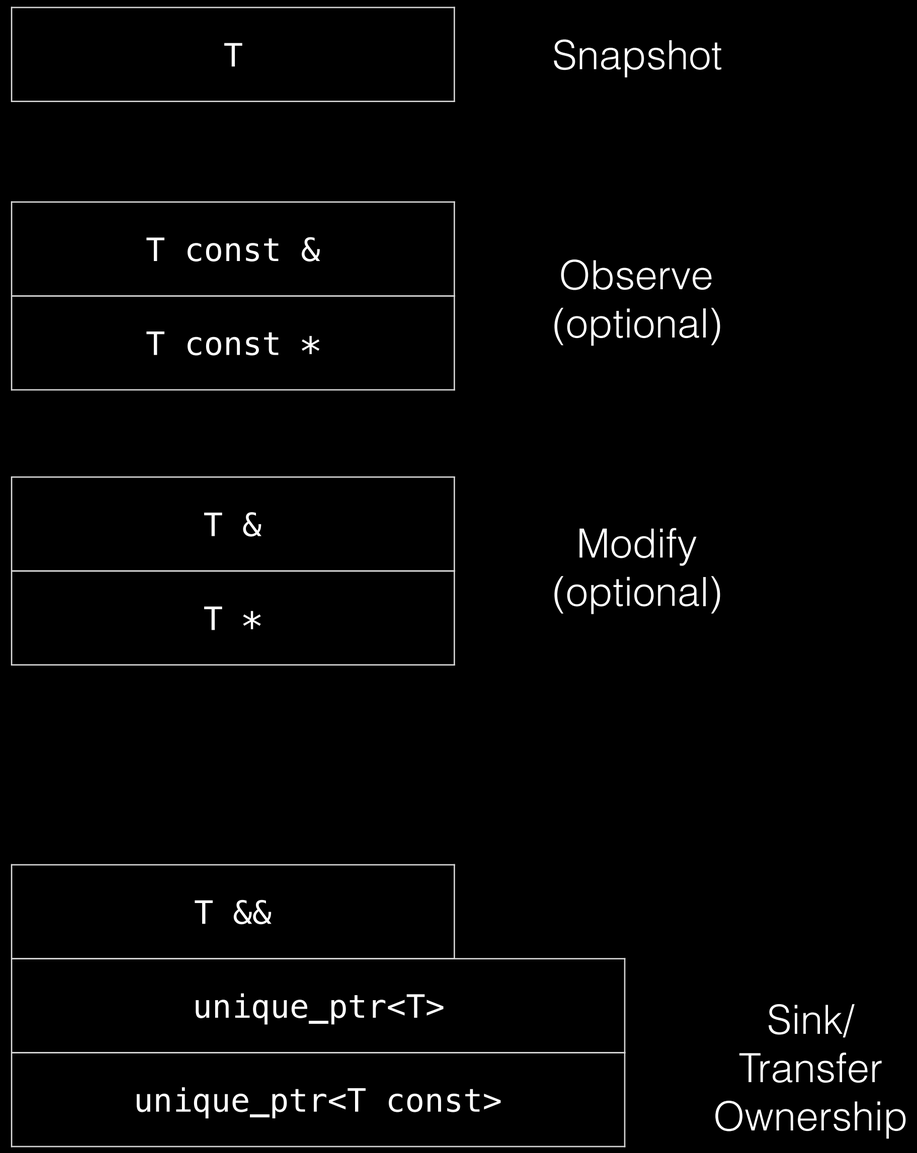
```
void f(???);
```

- Given `f()`, what are all the ways to pass arguments?

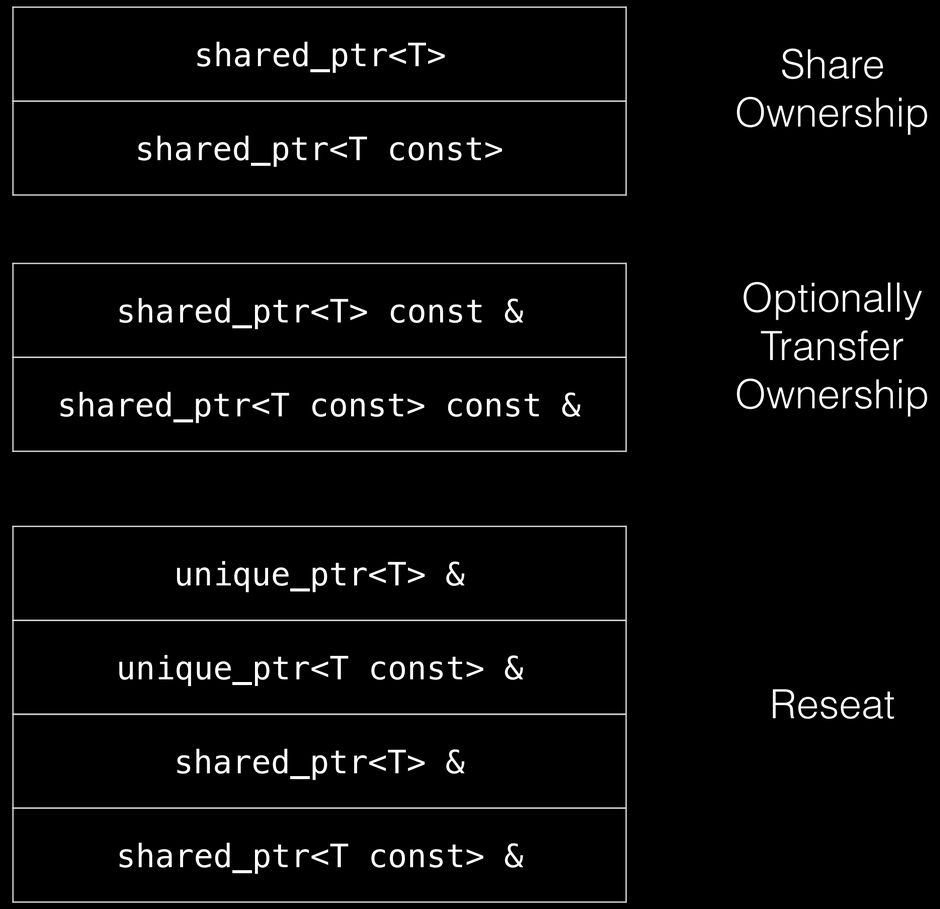
| | |
|---|---|
| `T` | |
| `T &` | `T const &` |
| `T &&` | |
| `T *` | |
| | |
| | |
| `T const *` | |
| | |
| | |

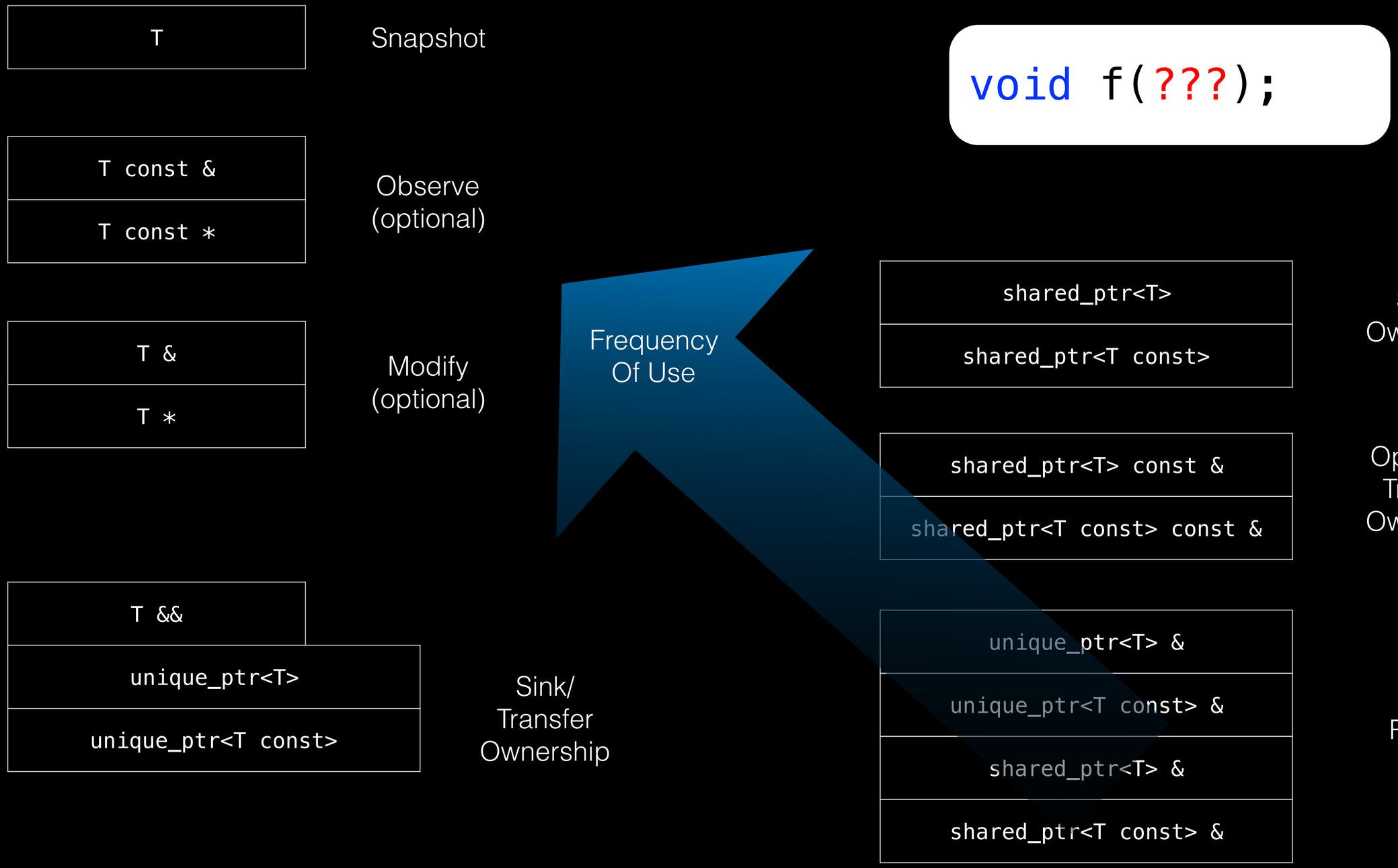
| | |
|---|---|
| `unique_ptr<T>` | |
| `unique_ptr<T> &` | |
| | |
| `unique_ptr<T const>` | |
| `unique_ptr<T const> &` | |
| | |

| | |
|---|---|
| `shared_ptr<T>` | |
| `shared_ptr<T> &` | `shared_ptr<T> const &` |
| | |
| `shared_ptr<T const>` | |
| `shared_ptr<T const> &` | `shared_ptr<T const> const &` |
| | |


T
Snapshot
T const &
T const *
Observe
(optional)
T &
T *
Modify
(optional)
T &&
unique_ptr<T>
unique_ptr<T const>
Sink/
Transfer
Ownership

```
void f(???);
```


shared_ptr<T>
shared_ptr<T const>
Share
Ownership
shared_ptr<T> const &
shared_ptr<T const> const &
Optionally
Transfer
Ownership
unique_ptr<T> &
unique_ptr<T const> &
shared_ptr<T> &
shared_ptr<T const> &
Reseat


T
Snapshot
void f(???);
T const &
T const *
Observe
(optional)
T &
T *
Modify
(optional)
Frequency
Of Use
shared_ptr<T>
shared_ptr<T const>
Share
Ownership
shared_ptr<T> const &
shared_ptr<T const> const &
Optionally
Transfer
Ownership
T &&
unique_ptr<T>
unique_ptr<T const>
Sink/
Transfer
Ownership
unique_ptr<T> &
unique_ptr<T const> &
shared_ptr<T> &
shared_ptr<T const> &
Reseat

# Avoid “Surprising” Sharing

```
void foo(Bar& input);



void some_work()
{
  Bar my_b;
  foo(my_b);
}
```

# Avoid “Surprising” Sharing

```
class Oops
{
  Bar& b;
  // more "stuff"
};
std::unique_ptr<Oops> global;

void foo(Bar& input)
{
  global = std::make_unique(input);
}

void some_work()
{
  Bar my_b;
  foo(my_b);
} <<< Oops!!!
```

- “Surpising” sharing occurs when reference or pointer used outside of function execution.

# Avoid “Surprising” Sharing

```
class Oops
{
  std::weak_ptr<Bar> b;
  // more "stuff"
};
std::unique_ptr<Oops> global;

void foo(std::shared_ptr<Bar> input)
{
  global = std::make_unique(input);
}

void some_work()
{
  auto my_b = std::make_shared<Bar>();
  foo(my_b);
}
```

- Use `shared_ptr` and `weak_ptr` to express lifetime.

Will not used outside of function call

```
void foo(T *);
```

```
void foo(T &);
```

```
void foo(T const *);
```

```
void foo(T const &);
```

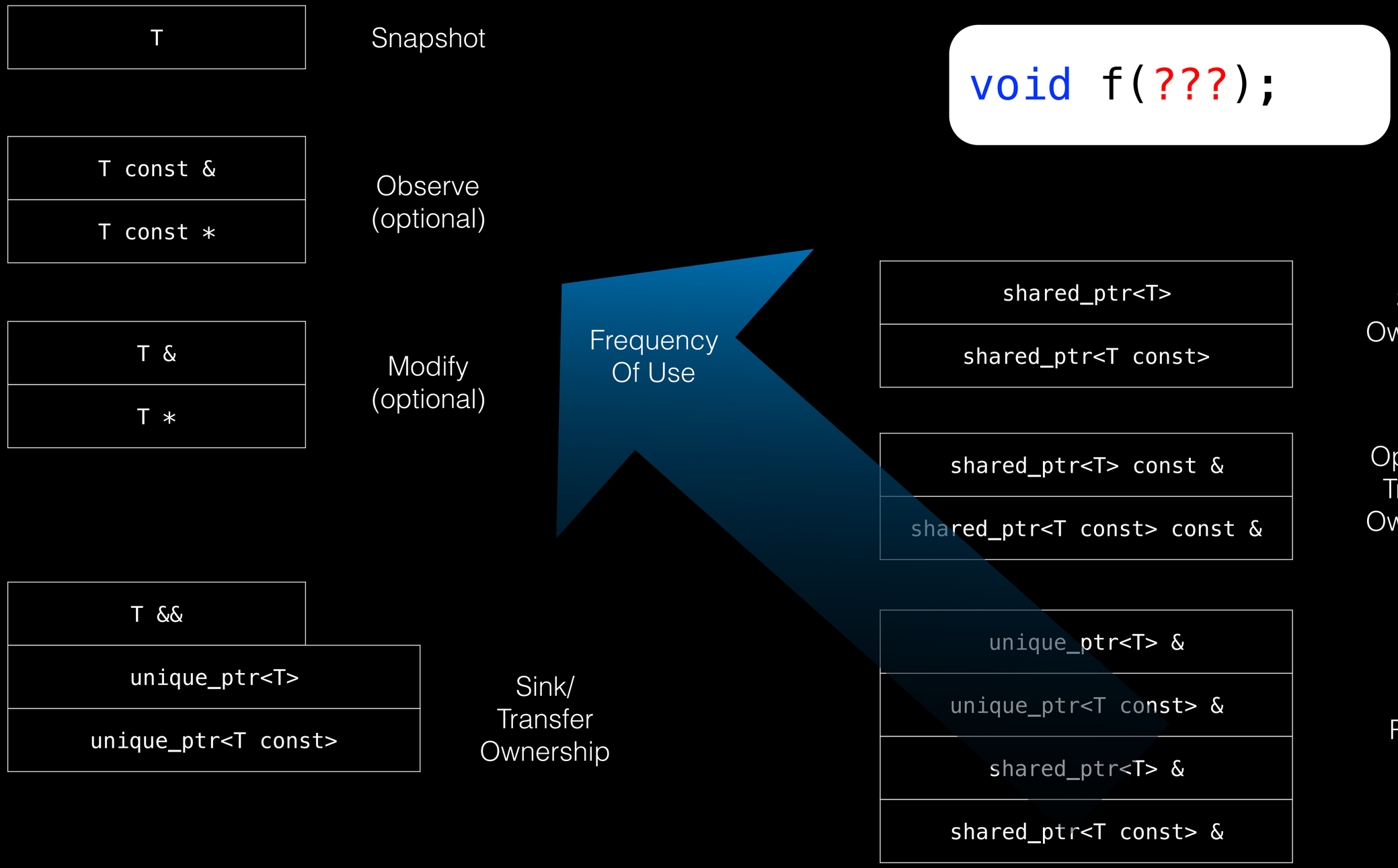
Will be used outside of function call

```
void foo(std::shared_ptr<T>);
```

```
void foo(std::shared_ptr<T const>);
```

# Guidelines

- Understand what your argument types mean and what you are saying.
- Pass by Value when you can, pass by `const` reference when you must.
- Use raw pointers for optional, `unique_ptr`/`shared_ptr` for lifetime.
- Return results.
- Don’t share via `*` or `&`, express via `shared_ptr`.


T
Snapshot
void f(???);
T const &
T const *
Observe
(optional)
T &
T *
Modify
(optional)
Frequency
Of Use
shared_ptr<T>
shared_ptr<T const>
Share
Ownership
shared_ptr<T> const &
shared_ptr<T const> const &
Optionally
Transfer
Ownership
T &&
unique_ptr<T>
unique_ptr<T const>
Sink/
Transfer
Ownership
unique_ptr<T> &
unique_ptr<T const> &
shared_ptr<T> &
shared_ptr<T const> &
Reseat