What do you *mean* "thread-safe"?

[Geoff Romer]

What does "thread-safe" mean?

"A thread-safe function can be safely invoked concurrently with other calls to the same function, or with calls to any other thread-safe functions, by multiple threads."

— POSIX

What does "thread-safe" mean?

```
int in[100], out[100];
```

```
void thread1() {
  …
  memcpy(&out, &in,
         sizeof(in));
  …
}
```

```
void thread2() {
  …
  memcpy(&out, &in,
         sizeof(in));
  …
}
```

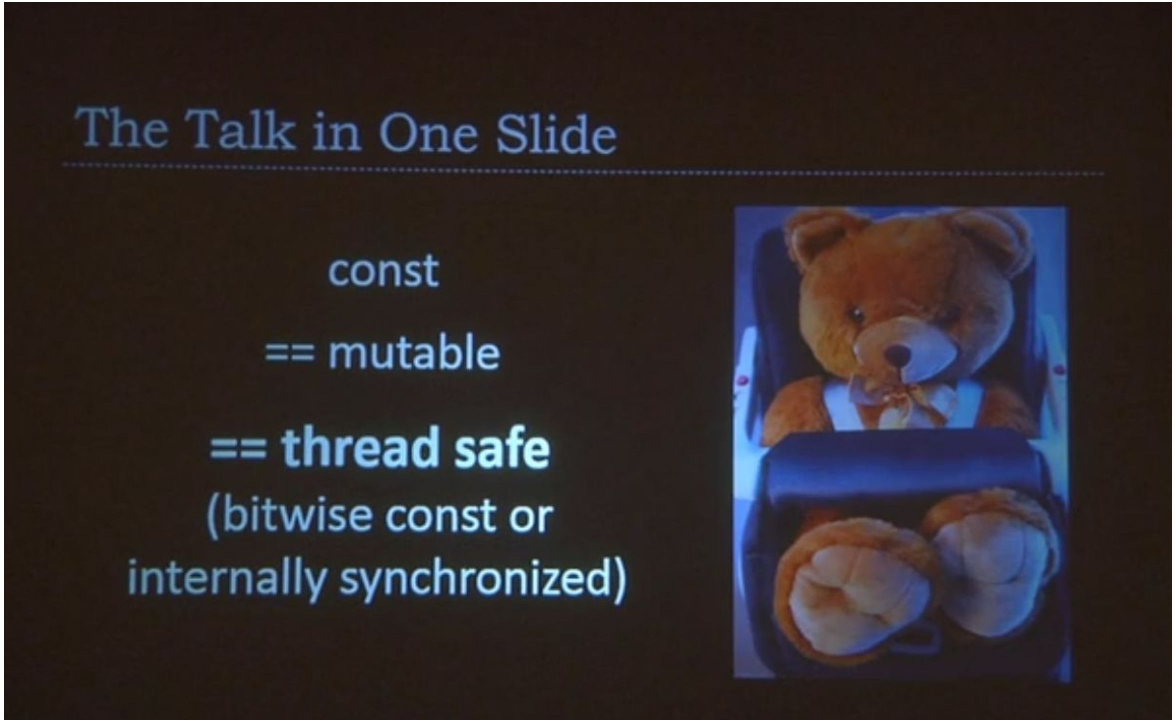
What does "thread-safe" mean?

```
int in[100];

void thread1() {
  …
  int out[100];
  memcpy(&out, &in,
         sizeof(in));
  …
}
```

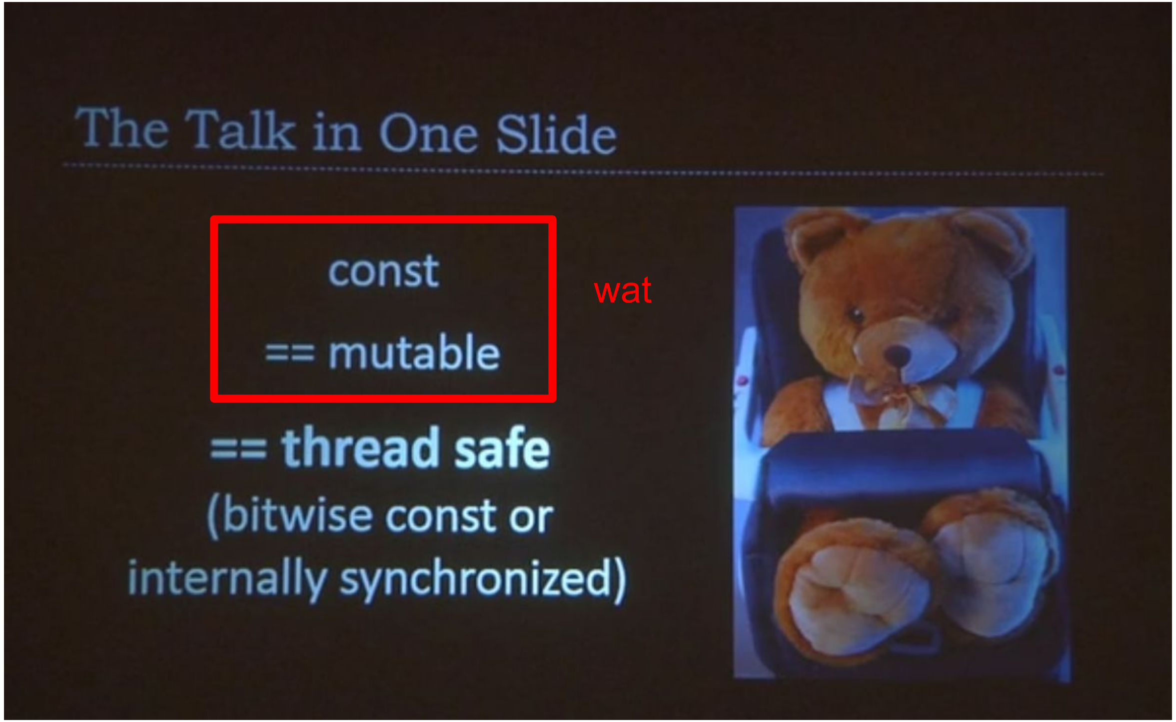
```
void thread2() {
  …
  int out[100];
  memcpy(&out, &in,
         sizeof(in));
  …
}
```

What does "thread-safe" mean?



— Herb Sutter

What does "thread-safe" mean?



— Herb Sutter

What are we trying to be "safe" from?

What are we trying to be "safe" from?

Race conditions?

"A race condition or race hazard is the behavior of an electronics, software, or other system where the **output is dependent on the sequence or timing** of other uncontrollable events." — Wikipedia

What are we trying to be "safe" from?

Data races?

```
int i = 0;
```

```
void thread1() {
  …
  ++i;
  …
}
```

```
void thread2() {
  …
  std::cout << i;
  …
}
```

What are we trying to be "safe" from?

Data races?

```
std::string s = "";
```

```
void thread1() {
  …
  s.append("foo");
  …
}
```

```
void thread2() {
  …
  std::cout << s;
  …
}
```

What are we trying to be "safe" from?

Data races?

"The execution of a program contains a *data race* if it contains two potentially concurrent conflicting actions, at least one of which is not atomic...

Two expression evaluations *conflict* if one of them modifies a memory location (4.4) and the other one reads or modifies the same memory location."

— The C++ Standard

What are we trying to be "safe" from?

Data races?

```
std::string s = "";
```

```
void thread1() {
  …
  s.append("foo");
  …
}
```

```
void thread2() {
  …
  std::cout << s;
  …
}
```

What are we trying to be "safe" from?

API races

An *API race* occurs when the program performs two concurrent operations on the same object, when the object's API contract doesn't permit those operations to be concurrent.

Identifying API races

```
Widget shared_widget;

void thread1() {
  …
  shared_widget.foo();
  …
}

void thread2() {
  …
  shared_widget.bar();
  …
}
```

Identifying API races

```
Widget shared_widget;

void thread1() {
  Thingy t;
  …
  t.foo(shared_widget);
  …
}
```

```
void thread2() {
  Whatever w;
  …
  w.bar(shared_widget);
  …
}
```

Identifying API races

If a live object has a **thread-safe type**, it can't be the site of an API race.

Identifying API races

```
// Thread-safe
class JobRunner {
  JobSet running_;
  JobSet done_;
  std::mutex m_;

  void OnJobDone(Job* job) {
    m_.lock();
    running_.erase(job);
    done_.insert(job);
    m_.unlock();
  }
};
```

```
// Thread-safe
class JobSet {
  std::set<Job*> jobs_;
  std::mutex m_;

  void erase(Job* job) {
    m_.lock();
    jobs_.erase(job);
    m_.unlock();
  }
};
```

Identifying API races

```
int shared_int;

void thread1() {
  Thingy t;
  …
  t.foo(shared_int);
  …
}

void Thingy::foo(int i);
```

```
void thread2() {
  Whatever w;
  …
  w.bar(shared_int);
  …
}

void Whatever::bar(
    const int& i);
```

Identifying API races

```
Widget shared_widget;

void thread1() {
  Thingy t;
  …
  t.foo(shared_widget);
  …
}

void Thingy::foo(
    const Widget& widget)
```

```
void thread2() {
  Whatever w;
  …
  w.bar(shared_widget);
  …
}

void Whatever::bar(
    const Widget& widget);
```

Identifying API races

If a live object has a **thread-safe type**, it can't be the site of an API race.

If a live object has a **thread-compatible type**, it can't be the site of an API race if it's not being mutated.

Identifying API races

```
class LazyStringView {
  const char* data_;
  mutable optional<size_t> size_;
  mutable std::mutex mu_;

 public:
  size_t size() const {
    std::scoped_lock lock(mu_);
    if (!size_)
      size_ = strlen(data_);
    return *size_;
  }
};
```

Identifying API races

```
Widget shared_widget;

void thread1() {
  Thingy t;
  …
  t.foo(shared_widget);
  …
}

void thread2() {
  Whatever w;
  …
  w.bar(shared_widget);
  …
}
```

Identifying API races

If a live object has a **thread-safe type**, it can't be the site of an API race.

If a live object has a **thread-compatible type**, it can't be the site of an API race if it's not being mutated.

If a live object has **any type**, it can't be the site of an API race if it's not being accessed concurrently.

Identifying API races

```
struct Counter {
  int c = 0;
  void operator()() { ++c; }
};
const std::function<void()> f = Counter{};
```

```
void thread1() {
  f();
}
```

```
void thread2() {
  f();
}
```

Identifying API races

```
Widget shared_widget;

void thread1() {
  Thingy t;
  …
  t.foo(shared_widget);
  …
}
```

```
void thread2() {
  Whatever w;
  …
  w.bar(shared_widget);
  …
}
```

Identifying API races

```
void Thingy::foo(
    const Widget&) {
  …
  baz();
  …
}

void baz() {
  static int counter = 0;
  counter++;
}
```

```
void Whatever::bar(
    const Widget&) {
  …
  baz();
  …
}
```

Identifying API races

If a live object has a **thread-safe type**, it can't be the site of an API race.

If a live object has a **thread-compatible type**, it can't be the site of an API race if it's not being mutated.

If a live object has **any type**, it can't be the site of an API race if it's not being accessed concurrently.

... but beware of **thread-hostile functions**, which can cause API races at sites other than their inputs.

Identifying API races

A given line of code is guaranteed to have no API races if it calls no thread-hostile functions, all inputs are live, and each input is

- not being accessed by other threads, or
- thread-safe, or
- thread-compatible and not being mutated by any thread.

Identifying API races

```
vector<int> shared_vec = {0, 0};

void thread1() {
  …
  ++shared_vec[0];
  …
}

void thread2() {
  …
  ++shared_vec[1];
  …
}
```

Identifying API races

```
vector<bool> shared_vec = {false, false};

void thread1() {
  …
  shared_vec[0] = true;
  …
}

void thread2() {
  …
  shared_vec[1] = true;
  …
}
```

Identifying API races

```
vector<int> shared_vec = {0, 0};

void thread1() {
  …
  ++shared_vec[0];
  …
}

void thread2() {
  …
  ++shared_vec[1];
  …
}
```

Identifying API races

```
// Increments every value in the range [begin, end).
template <typename Iterator>
void f(Iterator begin, Iterator end) {
  for (Iterator it = begin; it != end; ++it)
    ++*it;
}

std::vector<int> v = {1, 2, 3};
```

```
void thread1() {
  f(v.begin(),
    v.begin() + 2);
}
```

```
void thread2() {
  f(v.begin() + 1,
    v.end());
}
```

Identifying API races

```
class Widget {
  int* counter_;

 public:
  Widget(int* counter)
    : counter_(counter) {}

  // Thread-hostile!
  void Twiddle() {
    ++*counter_;
  }
};
```

```
Widget MakeWidget();

void thread1() {
  Widget w = MakeWidget();
  w.Twiddle();
}

void thread2() {
  Widget w = MakeWidget();
  w.Twiddle();
}
```

Recommendations

For library code:

- Make your types thread-compatible if possible, thread-safe if necessary.
- Clearly document any types that are thread-safe, or thread-incompatible.
    - Prefer to explicitly document the rest as thread-compatible.
- Be thoughtful about directly exposing sub-objects.
- Never define or use thread-hostile functions.
    - Avoid hidden mutable shared state
    - Be very careful with private pointers to shared data.

For application code:

- Make shared objects thread-safe, or thread-compatible and immutable.

Questions?